Resumen:

En este trabajo de fin de grado se ha llevado a cabo el diseño una aplicación basada en lenguaje Python que permite establecer una comunicación por Radio Frecuencia (RF) entre un microcontrolador (Pyboard con MicroPython) y un microcomputador (Raspberry Pi). Este trabajo es un primer paso que posteriormente permitirá la comunicación entre nuestro microcomputador y otro sistema autónomo, como puede ser un robot terrestre o un UAV (*unmaned aerial vehicle*), remotamente, de forma segura y eficiente, como cualquier otro sistema de radio control (RC).

En las primeras fases del proyecto se realiza un análisis de las características del hardware con el que se realizará la aplicación. Entre ellos se encuentran el microcontrolador y el microcomputador, pero también los módulos de RF XBee Pro 60mW - Series 1 (802.15.4) y la unidad de movimiento inercial o *inertial movement unit* (IMU) MPU6050, que permitirá al microcontrolador conocer su configuración en el espacio. Además, también se hará una breve explicación de todo el hardware complementario que permitirá el correcto funcionamiento del sistema.

Posteriormente se crearán las funciones para establecer la comunicación entre los dos dispositivos con los módulos XBee, por medio de un periférico *universal asincronous receiver transmiter* (UART). Estos módulos permitirán el envío serie asíncrono de paquetes de datos, ya sean las órdenes que enviemos desde la Raspberry Pi (RPi) o la información que recopile el microcontrolador. Nuestros dispositivos serán receptores (RX) y transmisores (TX) simultáneamente, de manera que se considerará que nuestra aplicación trabaja en modo full dúplex.

Para terminar el desarrollo se ha diseñado una interfaz gráfica muy sencilla que permitirá al usuario tener una pequeña realimentación (*feedback)* de lo que está pasando entre los dispositivos e interactuar con el sistema autónomo. Finalmente se ha realizado un test de validación para comprobar que la aplicación funciona correctamente y cumple los objetivos planteados.

Índice:

Índice de Figuras:

Índice de Tablas:

1. Introducción

El objetivo de este proyecto es el desarrollo de una aplicación para hacer posible la comunicación de un microcomputador (Raspberry Pi) y un sistema autónomo con un microcontrolador mediante RF basado en un módulo XBee Pro 60mW - Series 1 (802.15.4) para la posterior introducción en un sistema RC.

Para el desarrollo de este proyecto se ha contado con el apoyo del Departamento de Ingeniería Electrónica de la ETSEIB y gracias a la asignatura Desarrollo de aplicaciones basadas en microcontroladores que se ha impartido en el último curso del Grado de Ingeniería en Tecnologías Industriales, lo que ha permitido que este proyecto diera luz.

1.1 Objetivos del proyecto

- Establecer una comunicación inalámbrica fiable entre dos dispositivos mediante el módulo de comunicación XBee Pro 60mW - Series 1 (802.15.4), y que ésta permita el intercambio de datos entre los dos dispositivos correctamente.

- Desarrollar un programa en lenguaje Python para que ambos dispositivos inicien una comunicación RF y funcionen correctamente. También se dispondrá de una pequeña interfaz gráfica para que el operador o usuario reciba la información que envía el microcontrolador.

- Verificar que es una comunicación robusta y eficiente entre los dispositivos y elaborar un informe del proceso interno del microcontrolador (Pyboard con MicroPython).

1.2 Alcance del proyecto

El principal reto de este proyecto es la elaboración de un software propio adaptado a esta aplicación para que funcione correctamente. El software debe incorporar diferentes módulos de acuerdo con las necesidades de cada plataforma. Por una parte tendremos la Raspberry Pi y por otra el microcontrolador incorporado a la placa Pyboard utilizando MicroPython, que corre una versión del Python3.0 que optimiza el funcionamiento y permite un uso rápido y fácil del hardware.

La finalidad de este trabajo es, en definitiva, crear una herramienta de trabajo para los estudiantes de electrónica en sus prácticas de Radio Frecuencia, de modo que no se profundizará ni divagará en la gran variedad de usos prácticos y ventajas que puede suponer la comunicación inalámbrica entre microcontroladores.

En el momento en que se alcance una comunicación sencilla, versátil, eficiente y fiable entre diferentes microcontroladores/microcomputadores se dará por alcanzado el objetivo principal de este proyecto.

1.3 Objetivos personales

Durante el desarrollo de este proyecto han surgido problemas que no son propios del desarrollo de una aplicación concreta, sino que son comunes en cualquier proyecto en el que interviene la electrónica y la programación. Por esta razón este proyecto ha permitido poner en práctica conocimientos, adquirir experiencia de taller (soldar, cables, placas circuito impreso...) y aprender a solucionar problemas que aparecen cuando se programa una aplicación donde diferentes módulos tienen que colaborar conjuntamente.

2. Introducción al hardware

2.1 MicroPython y la Pyboard

El proyecto nació de un ingeniero australiano llamado Damien George que tenía el hobby de armar robots en su tiempo libre. Estaba buscando una nueva manera de programar sus creaciones, que fuera más sencilla que las alternativas que existían hasta el momento. Así pensó en Python, uno de los lenguajes de programación más populares en la actualidad. Sin embargo, Python requería demasiados recursos para las aplicaciones que él estaba buscando implementar, por lo que decidió reescribir completamente el lenguaje para hacerlo más liviano y eficiente, y así surgió MicroPython.

MicroPython es compilado a un código comprimido de bytes que corren en una máquina virtual, que dispone de algunas optimizaciones en velocidad. Tiene además un ensamblador en línea (*in-line)*, lo que podría en algún caso ayudar a escribir las partes críticas en código máquina, haciendo que los programas puedan correr aún más rápido.

Junto a este nuevo lenguaje, Damien George decidió crear también una placa para transformar su creación en un completo proyecto de software y hardware libre. La placa de MicroPython, la Pyboard (véase la Fig. 1), está basada en el microcontrolador ARM de 32 bits STM32F405, uno de los más potentes que hay disponibles y que puede aprovechar todas las posibilidades que el lenguaje de programación ofrece. En un pequeño tamaño de 3,3×4 centímetros, la placa que corre MicroPython funciona a una frecuencia de 168 MHz, tiene 1MB de memoria Flash y 192KB de memoria RAM y especificaciones técnicas suficientes para hacer funcionar correctamente MicroPython.

Se conecta al PC a través de un puerto USB y se puede interactuar con éste vía la

línea de comandos de Python. Puede compilar y correr *scripts* de Python sin ayuda del PC. Tiene un buen número de periféricos: 4 LEDs, un interruptor, reloj de tiempo real, acelerómetro y 30 pines de E/S. 5 UARTs, 2 SPIs, 2 canales I2C, 14 Pines ADC, 2 pines DAC y 4 pines PWM para el control de servomotores, contando además con un espacio para una tarjeta microSD para aumentar la capacidad de almacenamiento. Estas características permiten la conexión con el computador para cargar el código MicroPython y los elementos necesarios para realizar algunos experimentos, desde los más básicos hasta los más avanzados.

La placa Pyboard se soporta por el módulo *pyb* el cual incluye la función de encender LEDs, censar los interruptores, leer el acelerómetro, controlar los servos, etcétera. Además, dispone de funciones para el reloj de tiempo real, UART, I2C entre otras.

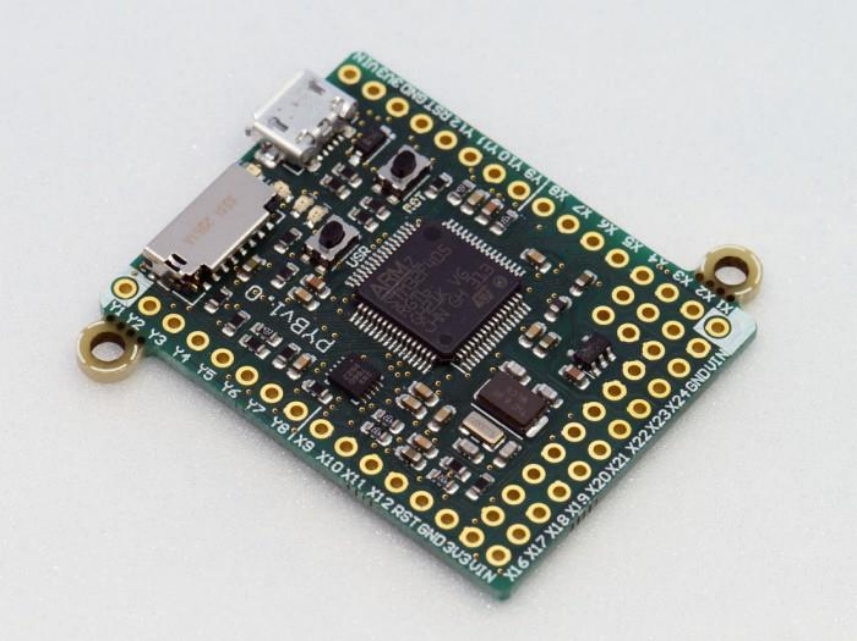


Figura 1. Placa Pyboard sobre la que funciona MicroPython

2.2 Raspberry Pi B+

El proyecto Raspberry Pi se inició en 2006, cuando Eben Upton y algunos compañeros de trabajo de la Universidad de Cambridge decidieron solucionar un problema muy básico: la mala (o falta de) educación que demostraban los aspirantes a ingeniero informático. Eben Upton estaba a cargo de conseguir nuevos y buenos estudiantes para la universidad, pero se dio cuenta de que la mayoría de ellos tenían poco o ningún conocimiento de programación. Pensó que una parte del problema podría ser que ninguno de ellos poseía un PC realmente apropiado para programar y probar cosas nuevas, de forma que empezó con el diseño del micro-ordenador que ahora es conocido como Raspberry Pi.

El hardware de la Raspberry Pi tiene unas dimensiones de placa de 8,5 por 5,3 cm, en el modelo B+ que es el que se utilizará en este proyecto (véase Fig. 2), y cuenta con unas características muy interesantes. En su corazón nos encontramos con un chip integrado Broadcom BCM2835 y un procesador SoC ARM1176JZF-S core a 700MHz con capacidad de funcionamiento a varias frecuencias y la posibilidad de incrementarla (overclocking) hasta 1 GHz, el procesador gráfico VideoCore IV que soporta hasta una resolución de 1920x1200, 512 MB de memoria RAM, una ranura para una tarjeta microSD, un puerto Ethernet, 4 puertos USB 2.0, HDMI, audio/video Jack y 40 pines de entrada/salida digital entre otros.

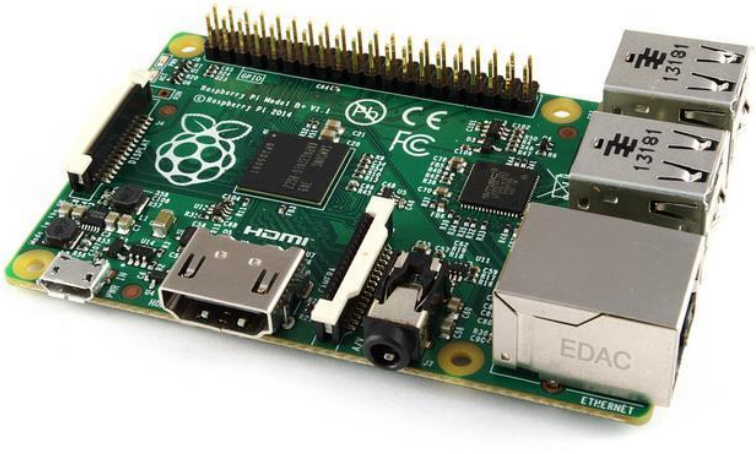


Figura 2. La Raspberry Pi modelo B+

2.3 XBee Pro 60mW - Series 1 (802.15.4)

Los módulos de comunicación inalámbrica XBee utilizados en este proyecto son económicos, potentes y fáciles de utilizar. Con forma de pequeños chips azules (véase Fig. 3), son capaces de comunicarse de forma inalámbrica unos con otros. Pueden hacer cosas simples, como reemplazar cables en una comunicación serie, lo cual facilita mucho la comunicación cuando deseas crear, por ejemplo, un vehículo de radio control.

De acuerdo con Digi, los módulos XBee son soluciones integradas que brindan un medio inalámbrico para la interconexión y comunicación entre dispositivos. Estos módulos utilizan el protocolo de red llamado IEEE 802.15.4 para crear redes FAST POINT-TO-MULTIPOINT (punto a multipunto); o para redes PEER-TO-PEER (punto a punto). Fueron diseñados para aplicaciones que requieren de un alto tráfico de datos, baja latencia y una sincronización de comunicación predecible, en términos simples, los XBee son módulos inalámbricos fáciles de usar. Los módulos XBee son propiedad de Digi basados en el protocolo Zigbee, los XBee Series 1 (también llamados XBee 802.15.4). Son la manera más fácil para trabajar, no necesitan ser

configurados, pero incluso así se pueden obtener beneficios. Debido a que son fáciles para trabajar, por ese motivo son altamente recomendables, especialmente si se está empezando en el ámbito de la electrónica inalámbrica. Para comunicaciones Punto-a-Punto, estos módulos trabajan tan bien como los de la Serie 2, pero sin todo el trabajo de configuración previo.

Algunas de sus principales características son:

- Buen Alcance: hasta 1 milla (1.6 Km) para los módulos XBee Pro.
- 9 entradas/salidas con entradas analógicas y digitales.
- Bajo consumo (<50mA) cuando están en funcionamiento y <10µA cuando están en modo de bajo consumo (sleep).
- Interfaz serie.
- 65,000 direcciones para cada uno de los 16 canales disponibles. Se pueden tener muchos de estos dispositivos en una misma red.
- Fáciles de integrar.

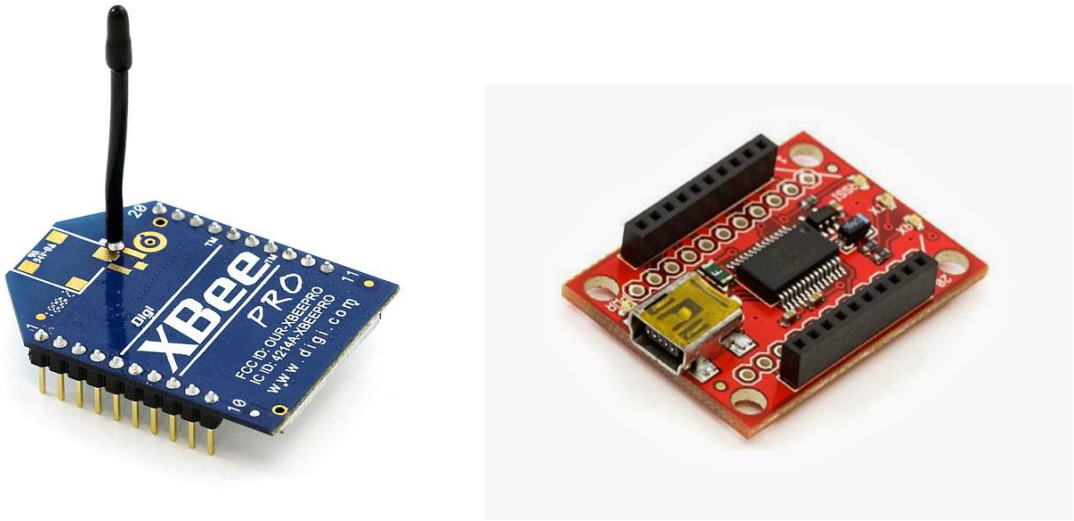


Figura 3. XBee pro Serie 1 y XBee explorer

Para la configuración previa de los módulos de RF se necesita una placa *XBee explorer* (véase Fig. 3) que se puede adquirir en la página web de *sparkfun electronics*, y mediante USB se conectará al ordenador, y al programa X-CTU (véase

Fig.4) que la propia empresa facilita gratuitamente en su página web. Es una interfaz muy sencilla, que permite la comunicación en modo transparente entre los dos módulos.

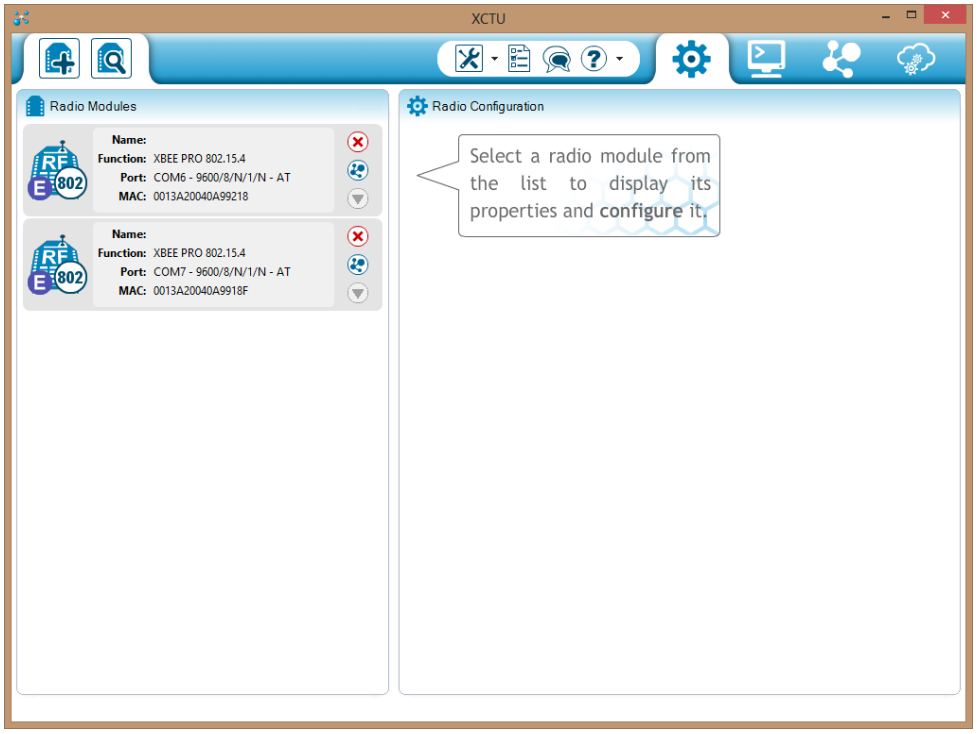


Figura 4. Programa X-CTU

En la comunicación en modo transparente (*peer-to-peer* o punto a punto) todo lo que ingresa por el Pin 3 (Data in), es guardado en el buffer de entrada y luego transmitido. Todo lo que ingresa como paquete RF, es guardado en el buffer de salida y luego enviado por el Pin 2 (Data out). El modo Transparente viene por defecto configurado en los módulos XBee. Este modo está destinado principalmente a la comunicación punto a punto, donde no es necesario ningún tipo de control. También se usa para reemplazar alguna conexión serie por cable, ya que es la configuración más sencilla posible y no requiere una configuración más avanzada.

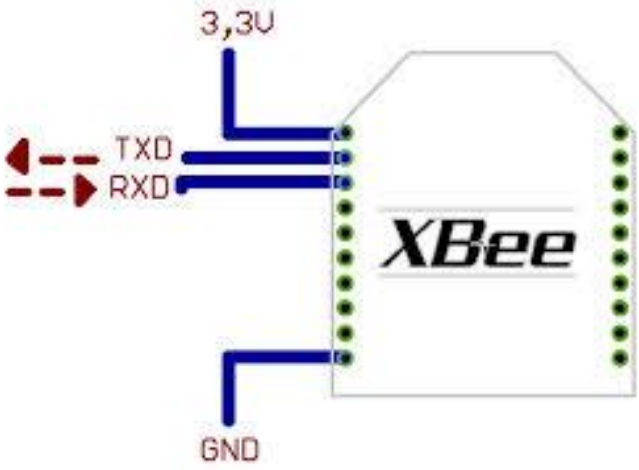


Figura 5. Esquema de pines XBee

2.4 MPU6050

El MPU6050, sensor de movimiento inercial, está basado en un circuito integrado de pequeñas dimensiones, apenas 25.5x15.2mm^2 (véase Fig. 6). Este pequeño chip actúa como sensor de movimiento (IMU) de hasta 6 grados de libertad oDOF (**"**6 Degrees Of Freedom"). Esto significa que lleva un acelerómetro y un giroscopio, ambos de 3 ejes (3 + 3 = 6 DOF) con una alta precisión. Posee ADC internos de 16 bits, lo cual significa que divide el rango dinámico en 65536 fracciones, y utiliza el protocolo de comunicación I2C para poderse comunicar con cualquier microcontrolador como el de la placa Pyboard, Arduino... El MPU-6050 opera con 3,3 voltios, aunque algunas versiones llevan un regulador que permite conectarlo a 5V.

Figura 6. MPU6050

Un acelerómetro mide la aceleración, inclinación o vibración y transforma dicha magnitud física en otra magnitud eléctrica que será la que se empleará en los equipos de adquisición estándar. Los rangos de medida van desde las décimas de *g*, hasta los miles de *g*.

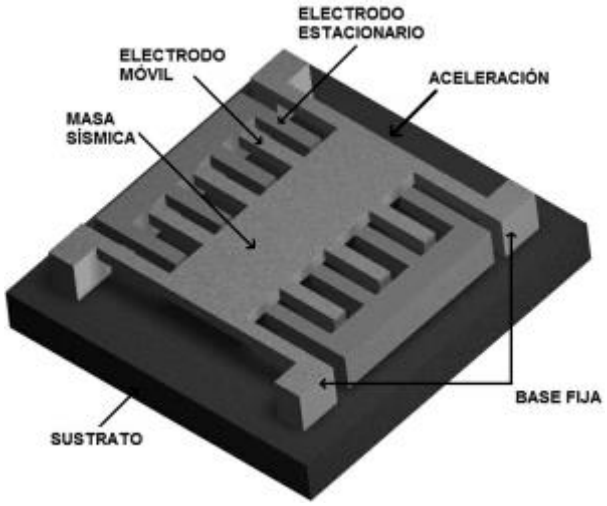


Figura 7. Funcionamiento del acelerómetro

Un giroscopio es un dispositivo que funciona para medir velocidades angulares basándose en el mantenimiento del impulso de rotación. Si se intenta hacer girar un objeto que está girando sobre un eje que no es el eje sobre el que está rotando, el objeto ejercerá un momento contrario al movimiento con el fin de preservar el impulso de rotación total. El giroscopio muestra el cambio de rango en rotación en sus ejes X, Y y Z.

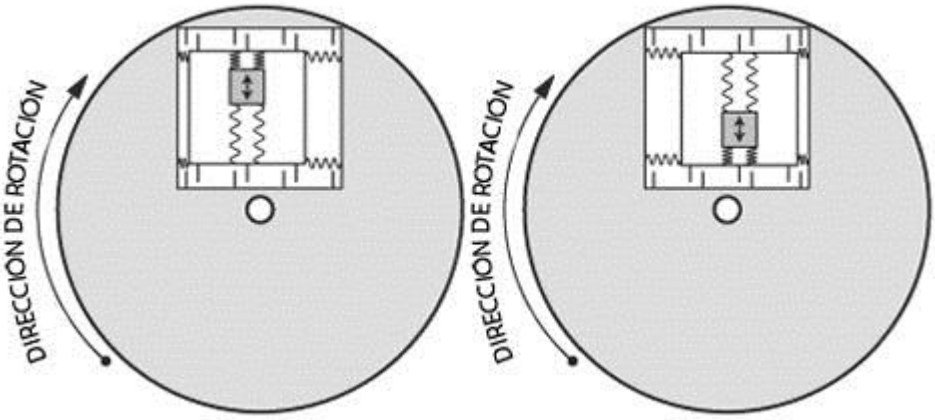


Figura 8. Funcionamiento del giroscopio

Una de las ventajas que cabe mencionar es, que el MPU-6050 posee conversores analógicos digitales para cada uno de los ejes de cada uno de los sensores, de manera que se pueden obtener los valores de forma simultánea con un rango de hasta 2000°/s para el giroscopio y hasta +/-16g para el acelerómetro.

3. MicroPython

Como se ha explicado en el capítulo de introducción al hardware, la placa Pyboard corre el MicroPython, una versión modificada de Python 3 adaptada para que pueda ejecutar el programa más rápido. En este apartado se hará una breve introducción a este pseudo-lenguaje con algunos ejemplos y se explicará cómo se ha probado el código al mismo tiempo que se iba escribiendo y desarrollando.

Primero de todo, para la programación y posterior puesta a punto del código, se necesita un ordenador con puertos de entrada USB libres. En estos puertos será donde se conectará la placa de MicroPython para cargar los archivos e intercambiar información en el modo emulador. En dicho modo se pueden controlar los pines del microcontrolador y usar el módulo pyb (se explicará más adelante). De esta forma permite al usuario probar los comandos y ver por el terminal y la placa la respuesta a las órdenes que se le han dado.

Otro elemento importante a tener en cuenta pero no separado del anterior es el sistema operativo que se va a usar para la realización de este proyecto. Es un factor muy importante ya que puede determinar en gran medida la complejidad del proyecto. De la página web oficial donde se describen los pasos para poder acceder al dispositivo se ha encontrado una gran diferencia en la complejidad de instalación entre los dos sistemas operativos más comunes (Windows y Ubuntu). En el sistema Windows se tienen que instalar los drivers manualmente y para entrar en el modo emulador se tiene que instalar otro programa para acceder a la consola. En cambio, en Ubuntu solo se tiene que conectar la placa por USB al ordenador y automáticamente detecta el dispositivo y se comporta como si fuera una memoria extraíble (cuando se tiene la tarjeta microSD) y se puede acceder directamente a los archivos. También en el modo emulador es mucho más sencillo. Únicamente se tiene que introducir el comando *sudo screen dev/ttyACM0.* Seguido de la contraseña que

pedirá y aparecerá un cursor como cuando se accede al intérprete de Python.

Teniendo en cuenta estos factores y otros de menor importancia (similitud entre Ubuntu y Raspbian…) se ha decidido utilizar el sistema operativo Ubuntu Desktop versión 14.04.

3.1 Primeros pasos

3.1.1 Módulo pyb

Cuando se accede por primera vez a la placa Pyboard con MicroPython se dispone de todos los módulos propios de Python pero además también lleva incorporado un módulo creado específicamente para este pequeño dispositivo. Este módulo adicional es el que permite controlar, por ejemple, los LEDs, y tiene funciones y clases predeterminadas que permiten controlar todos los periféricos que dispone la placa. Por ejemplo para los buses que se usarán (I2C y UART,) ya se dispone todos los protocolos que se necesitan. Estas funciones y clases ayudarán enormemente y facilitarán muchísimo la faena de programar, ya que el programador no se tendrá que preocupar del funcionamiento interno de la UART o del I2C.

3.1.2 Crear y editar el archivo main.py

Cuando se conecta el microcontrolador con la tarjeta microSD al ordenador por primera vez, éste instalará todos los componentes necesarios para su correcta utilización y cuando haya acabado se abrirá una ventana que mostrará todos los archivos que tiene guardados la tarjeta microSD. En esta ubicación se deberá crear un archivo con el nombre *main.py.* Éste será el archivo que se ejecute cuando se reinicie el microcontrolador.

Una vez creado, para acceder a él y editarlo se puede hacer de dos formas distintas: la primera y más fácil es acceder al dispositivo como una unidad de almacenamiento externo y hacer doble clic sobre el icono, pero también hay otra que a veces puede resultar más cómoda si se está acostumbrado a usar el terminal. Se puede acceder a esta ubicación mediante la ruta que tiene la tarjeta microSD, en este caso es */media/nombredelequipo/722C-6C41/*, pero hay que tener en cuenta que ésta es una tarjeta única y cada una tiene un nombre diferente (en este caso la id de la tarjeta es 722C-6C41).

3.1.3 Resetear la placa MicroPython

Para ejecutar el nuevo programa que se haya escrito es necesario hacer un reset del microcontrolador. Hay muchas maneras de hacerlo pero aquí se va a explicar la que aporta más seguridad y va a evitar posibles problemas futuros.

Una vez realizado el cambio en los archivos deseados se deberán guardar todos, ya que cuando se provoque el reset todos los cambios que no se guarden se van a perder. Una vez guardados, se deberá expulsar el dispositivo con seguridad. Este paso evitará posibles errores en la escritura de la unidad extraíble que podría corromper algunos archivos y que sean irrecuperables. Por este motivo, también se recomienda tener en el propio ordenador una carpeta a modo de copia de seguridad para ir guardando copias de los archivos que se usen, para en caso de perder los originales no perder todo el trabajo hecho.

Cuando en el ordenador aparezca el mensaje de que es seguro retirar el dispositivo ya podremos resetear la placa de MicroPython, pero hay un problema: cuando pulsemos el botón RST de la placa, ésta se reiniciará y al verse conectada al ordenador volverá a conectarse como unidad de disco y dejará el programa a media ejecución. Para este problema existen 2 soluciones: la primera y más recomendable,

cuando se está creando y perfeccionando el código, es no desconectar el microcontrolador y probarlo en modo emulador (se explicará en el siguiente punto). La otra opción es conectar el microcontrolador a una batería externa que solo le proporcione la alimentación. De este modo, como no estará conectada al ordenador la placa MicroPython, será libre para ejecutar el programa principal indefinidamente y sin interrupción. Por ello, esta opción solo se recomienda en la parte final del proyecto, cuando el código ya esté listo y a prueba de errores, ya que de esta forma no se puede saber lo que está pasando internamente en el microcontrolador.

3.1.4 Modo emulador o "REPL prompt"

Como se ha comentado anteriormente, una alternativa que simplifica la experimentación con comandos y la comprobación del código es el uso de un terminal. De esta manera se puede probar el funcionamiento de la aplicación sin necesidad de resetear la Pyboard. Para entrar en este modo solo se tiene que introducir el comando *sudo screen dev/ttyACM0* y seguido de la contraseña que pedirá como se comentó con anterioridad. Una vez introducido aparecerá un terminal idéntico al intérprete de Python, pero con una única diferencia, este es el de la placa MicroPython, y por lo tanto todo lo que hagamos en esta consola tendrá su efecto en el microcontrolador, por ejemplo, si decidimos encender un led (como se verá en unos ejemplos breves a continuación), en nuestro microcontrolador también se encenderá. Aquí un pequeño ejemplo:

```
>>> pyb.LED(1).on()

>>> pyb.LED(2).on()
>>> 1 + 2
3
>>> 1 / 2
0.5
>>> 20 * 'py'
'pypypypypypypypypypypypypypypypypypypypy'
```

3.1.5 Soft reset

```
>>>
PYB: sync filesystems
PYB: soft reboot
Micro Python v1.0 on 2014-05-03; PYBv1.0 with STM32F405RG
Type "help()" for more information.
>>>
```

El “soft reset” es un reset del emulador. Presionando Ctrl+D, la consola vuelve a cargar todo el programa guardado en el *main.py*. Esto es extremadamente útil cuando se está en proceso de creación del código, ya que permite probar el código casi al instante sin tener que desconectar el microcontrolador ni tener que cerrar los editores de texto. Además, la consola muestra los posibles errores de complicación en el código.

3.2 Ejemplos de la documentación oficial:

Aquí se mostraran unos pequeños ejemplos de comandos escritos en la consola. Hay que tener en cuenta que cada vez que se entra en el modo emulador o se hace un “soft reset” se tiene que volver a importar el módulo de la Pyboard (`import pyb`)

3.2.1 LEDs

A continuación se muestran unos pequeños ejemplos de lo que se puede hacer en el modo emulador con unas pocas líneas de código. En estos fragmentos se encenderán y apagarán algunos de los leds de microcontrolador desde el terminal del ordenador. En el primero se encenderán y apagaran manualmente.

```
>>> myled = pyb.LED(1) # red led
>>> myled.on()
>>> myled.off()
```

En este otro fragmento se generará un bucle infinito que apagá y encenderá un led en intervalos de 1 segundo (1000 milisegundos). Una vez que se desee terminar el proceso se deberá presionar Ctrl+C para generar una interrupción voluntaria.

```
>>> led = pyb.LED(2) # green led
>>> while True:
...     led.toggle()
...     pyb.delay(1000)
...
```

Por último, existe un LED especial (el número 4 de color azul) que tiene la capacidad de regular su intensidad. Ésta se puede controlar mediante el método *.intensity(n)*, donde n es un número entero comprendido entre el 0 y el 255. En el siguiente ejemplo se hará que el led de color azul incremente su brillo gradualmente y cuando llegue a su máximo, se apague.

```
>>> led = pyb.LED(4) # blue led
>>> intensity = 0
>>> while True:
...     intensity = (intensity + 1) % 255
...     led.intensity(intensity)
...     pyb.delay(20)
...
```

3.2.2 El pulsador USR

La placa Pyboard de MicroPython dispone de dos botones, uno es el de reset (RST) que se ha comentado con anterioridad, pero dispone de otro pulsador que permite una interacción rápida con el usuario sin tener que añadir ningún hardware adicional. Este botón es el llamado de usuario (USR) y está situado justo delante de la ranura

de la tarjeta microSD. A continuación se mostraran unos pequeños ejemplos del uso de esta herramienta.

```
>>> sw = pyb.Switch()
>>> sw() # switch unpressed
False
>>> sw() # switch pressed
True
```

En el siguiente ejemplo se puede ver el resultado del pulsador cuando está apretado y cuando no lo está. Este botón será de gran ayuda cuando se quiera detener la aplicación sin la necesidad de un “hard reset”, que a su vez haría que ésta volviera a empezar. Por lo tanto, en este caso, se usará como una herramienta para detener el proceso y dejar el microcontrolador en modo “stand by”.

3.2.3 El acelerómetro

La placa Pyboard dispone de un acelerómetro interno solidario al circuito impreso y, aún que en este proyecto se usará un acelerómetro externo conectado por I2C, es un ejemplo que puede ayudar a entender el funcionamiento de un acelerómetro.

En este caso el acelerómetro del microcontrolador puede ayudar al movimiento de la placa y la orientación cuando está quieta, ya que cuando está en movimiento puede haber aceleraciones que no son debidas a la gravedad, por lo que el posicionamiento y orientación pueden ser incorrectos. Cuando el microcontrolador está quieto y, mediante funciones trigonométricas (arcsin, arccos …) se puede conocer los ángulos que tiene el microcontrolador respecto a unos ejes de referencia.

```
>>> accel = pyb.Accel()

>>> accel.x()
```

En este ejemplo se ha creado una clase acelerómetro y con el método *.x()* retorna un valor de aceleración en el eje x. Este valor es un entero que va entre -30 y 30. También se debe tener en cuenta que la mayoría de mediciones tienen un pequeño error o ruido debido a interferencias electromagnéticas, por lo que es recomendable el uso de filtros.

```
>>> accel = pyb.Accel()
>>> light = pyb.LED(3)
>>> SENSITIVITY = 3
>>> while True:
...     x = accel.x()
...     if abs(x) > SENSITIVITY:
            light.on()
```

En este ejemplo se crea una pequeña rutina que enciende y apaga un LED en función de la inclinación del microcontrolador. Este ejemplo se podría ampliar con todos los demás ejes, pero sería complicar el ejemplo y tampoco es el objetivo de este proyecto.

3.3 Código de la aplicación en MicroPython:

Una vez acabada la introducción al lenguaje MicroPython y los primeros pasos para familiarizarse con el microcontrolador se pasa a describir el cuerpo de la aplicación de este proyecto. En este caso como se ha comentado con anterioridad, se creará un sistema autónomo y por ello necesitará alimentación externa y el microcontrolador no estará conectado al ordenador. Por este motivo será necesario un pulsador para detener la aplicación en caso que sea necesario, tal y como se ha comentado en los apartados anteriores.

Primero de todo se tiene que pensar en la estructura que tendrá el código. En este proyecto se ha considerado la opción de crear el programa en un fichero aparte y desde el *main.py* importarlo y ejecutarlo, pero finalmente se ha decidido ejecutar el programa directamente en el fichero *main.py.* La estructura que se ha creído más conveniente y adecuada es: primero importar todos los módulos necesarios para el correcto funcionamiento y posteriormente proceder con la inicialización de todos los objetos que se usarán en la comunicación RF que posteriormente se comentarán con profundidad cada uno.

En este punto también se ha querido destacar que en el archivo *main.py* solo se implementará la estructura principal del programa, para que sea clara y lo más breve posible y todas las funciones complementarias serán llamadas, ya que estarán en diferentes ficheros. De esta manera, se mantiene una vista clara y estructurada del bucle principal y en caso de que se quiera trabajar en alguno de los módulos complementarios, éstos organizados según la función que realizarán.

Durante el transcurso de la creación del código se ha decidido incorporar un pequeño reporte de lo que está haciendo la aplicación que se guardará en la tarjeta microSD. De esta forma, se podrá acceder a un archivo llamado *report.txt*, que aportará información de cada iteración que ejecute la aplicación con un detalle de todo lo que envía y todo lo que recibe. Esto ayudará en la fase de puesta a punto y calibración de la aplicación.

3.3.1 Archivo main.py

Como se ha comentado con anterioridad, éste es el archivo principal de la aplicación, ya que será el que el microcontrolador ejecute primero cuando éste se reinicie. Este archivo dispondrá de 4 partes diferenciadas, siendo la primera la que realizará la importación de todos los módulos necesarios para que el programa principal se

ejecute.

```
import pyb
import imu
from pyb import I2C
from pyb import UART
import aux
#-----
pyb.LED(1).off()
pyb.LED(2).off()
pyb.LED(3).off()
pyb.LED(4).off()
#-----
text=aux.creatext()+'\n'
blnk=pyb.LED(2)
bot=pyb.Switch()
text+='leds\tok\n'
```

Como se ve en este fragmento del código que corresponde a la primera fase, se puede ver claramente al inicio cómo se importan todos los módulos necesarios. Primero de todo está la sentencia *import pyb.* Este módulo es muy importante, como se ha comentado con anterioridad, ya que incluye muchas funciones del microcontrolador. Posteriormente se han importado los archivos *aux* e *imu* que tal y como se comentará más adelante funciones de los buses UART e I2C. También se puede ver que se ha importado del módulo *pyb* los atributos I2C y UART, ya que son importantes en este proyecto y se ha decidido importarlos por separado.

Justo a continuación se le dará la ordena l microcontrolador de que apague los LEDs. Ésta es una medida que se ha adoptado para los posibles casos de que el programa principal de la última ejecución tuviera un error y alguno de los LEDs quedara encendido. De esta forma se asegura que todos los LEDs estarán apagados. Después, y con los últimos comandos de esta primera fase, se crea la variable que servirá para almacenar toda la información que recopila el microcontrolador y que se guardará como reporte en la tarjeta microSD. Esta variable se inicializa mediante una función guardada en otro fichero. Este fichero complementará las funciones del

programa principal, pero en esta variable solo creará la cabecera del reporte que se guardará en un archivo de texto. También se declararán los objetos del pulsador y un LED (número 2, de color verde), para poder ver que la aplicación se está ejecutando. Este LED verde parpadeará cada n iteraciones del bucle principal. Si este deja de parpadear, querrá decir que algún error ha sucedido.

En la segunda parte, a la que se le ha dado el nombre de inicialización, es donde se crearán todos los objetos de los buses de comunicación. En este caso solo se usarán 2 periféricos, un I2C y una UART, como se ve en la figura siguiente:

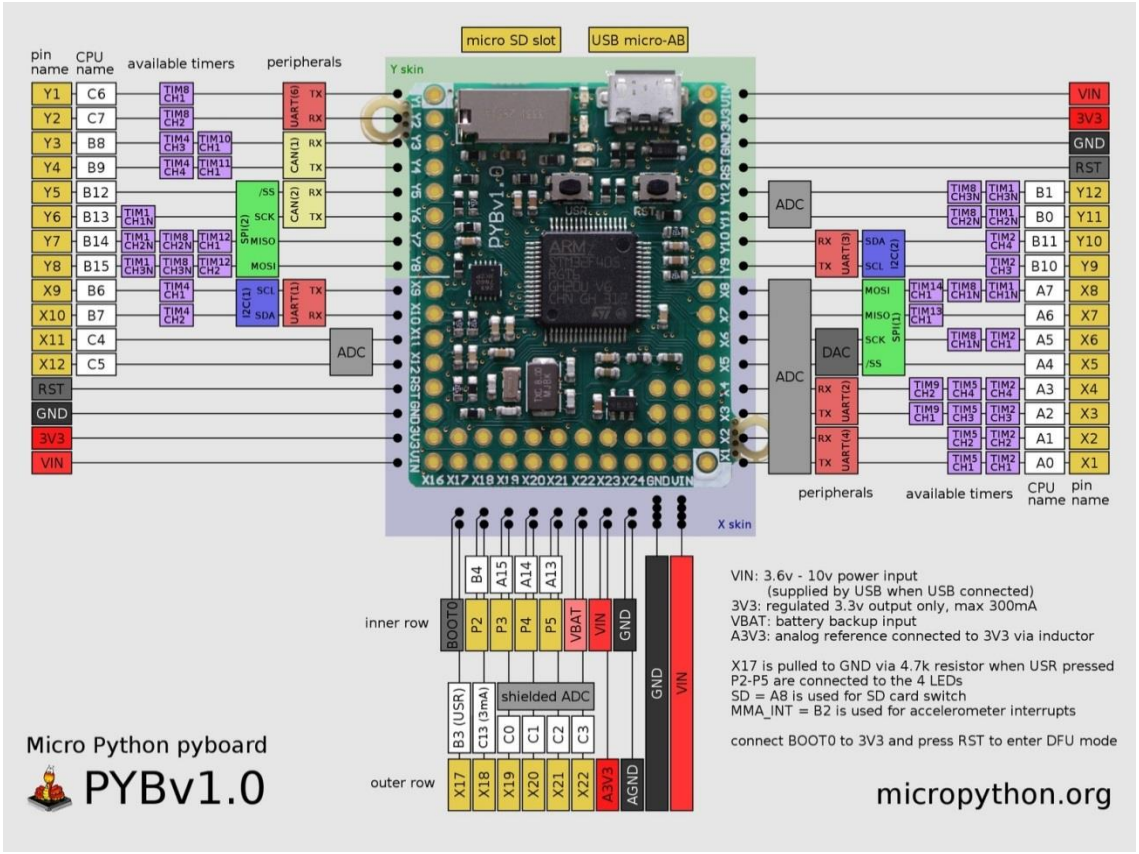


Figura 9. Esquema de la Pyboard

Se utilizará la UART conectada a los pines Y9 e Y10. Para la utilización de la UART se han usado los siguientes comandos:

```
uart=UART(3,9600)
uart.init(9600,bits=8,stop=1,timeout=10,parity=None)
```

En la primera línea se ve cómo se crea el objeto del periférico en los pines designados (en este caso se trata de la UART número 3 de las 5 posibles) y se le proporciona una velocidad o *baudrate*. Este parámetro será la velocidad en bits por segundo a la que se enviarán y recibirán los datos. En la segunda línea se inicializa el bus con información adicional que definirá sus características. Se trabaja con 8 bits de datos, 1 bit de stop y sin paridad. Finalmente, está el parámetro timeout, que se ha fijado en 10 ms.

En el caso del bus I2C se conectará en los pines X9 y X10 de la placa Pyboard. Éste bus no se comentará con mucha profundidad, ya que el objetivo del proyecto no es el control de un bus I2C. Por ello, se ha encontrado un código con licencia abierta del M.I.T para el control de una IMU (MPU9250) similar a la que se está usando en este caso y se han hecho unas pequeñas modificaciones para que sea compatible con la que se usa en este caso. Solo comentar que el microcontrolador actuará como *Maestro* y el dispositivo inercial actuará como *Esclavo* utilizando la dirección 0x68. Esta dirección permite que el bus I2C tenga más de dos dispositivos conectados y el *Maestro* pueda comunicarse con cada uno de ellos.

Para inicializar el bus I2C en el archivo principal se usa el siguiente comando:

```
i2c=imu.InvenSenseMPU(1,1,(0,1,2),(1,1,1))
```

Este comando llama a la función que inicializa el bus, pero ésta está en otro fichero, el *imu.py*, que es una modificación del programa creado por el M.I.T. En el pequeño fragmento que se ve a continuación pueden verse todos los pasos para la inicialización:

```
def __init__(self, side_str, device_addr, transposition, scaling):

    self._accel = Vector3d(transposition, scaling, self._accel_callback)
    self._gyro = Vector3d(transposition, scaling, self._gyro_callback)
    self.buf1 = bytearray([0]*1)            # Pre-allocated buffers for reads: allows reads to
    self.buf2 = bytearray([0]*2)            # be done in interrupt handlers
    self.buf3 = bytearray([0]*3)
    self.buf6 = bytearray([0]*6)
    self.mpu_addr = 0x68
    self.timeout = 5                       # I2C tieout mS

    tim = pyb.millis()                      # Ensure PSU and device have settled
    if tim < 200:
        pyb.delay(200-tim)

    self._mpu_i2c = pyb.I2C(1, pyb.I2C.MASTER)

    # Can communicate with chip. Set it up.
    self.wake()                             # wake it up
    self.passthrough = True                 # Enable mag access from main I2C bus
    self.accel_range = 0                    # default to highest sensitivity
    self.gyro_range = 0                     # Likewise for gyro
```

Posteriormente se abrirá el archivo *report.txt* en modo escritura y se guardará toda la información referente a la inicialización de la aplicación. Se hace de esta forma, grabando la información en la tarjeta muy a menudo, porque el microcontrolador dispone de una memoria RAM limitada y el hecho de ir almacenando información sin parar en una misma variable causaría un agotamiento de la memoria muy rápida. De esta forma, al ir guardándose periódicamente la información en la tarjeta microSD (que dispone de muchísima más capacidad de almacenamiento que la memoria RAM del dispositivo) y borrándose de la variable, se consigue no llenar toda la memoria del microcontrolador y no causar un error en la aplicación.

```
#guardar al fitxer la inicialitzacio
text+='\n#------------------#\n\n'
print(text)
f=open('report.txt','w')
f.write(text)
```

Una vez llegados a lo que sería el bucle principal del programa, la tercera de las partes del código, se tienen que declarar unas variables que serán de gran importancia en la ejecución del programa.

```
error=False
itera=0
cua=''
totaltram=[0,0]
```

Estas variables van a ser las más importantes en la ejecución del programa. Primero de todo se encuentra el booleano *error,* que será el encargado de parar el bucle si sucede cualquier fallo o si se decide interrumpir la aplicación voluntariamente. Seguido de esta variable va el contador de iteraciones. Esta variable será la encargada de decir cuántas iteraciones ha ejecutado la aplicación. Como elemento indispensable encontramos la variable *cua,* que será el lugar donde se almacenarán todos los datos que lleguen por radiofrecuencia al microcontrolador. Posteriormente se tratará, para sacar la información útil y procesarla. Más adelante se explicará el carácter de los datos que llegarán desde la Raspberry Pi y lo que significarán. Por último se ha declarado la variable *totaltram,* que es una lista que tiene dos parámetros. En ella se guardará el número de iteraciones que llegan a la Raspberry Pi correctamente y las que tienen errores. Estos valores se obtendrán del post-procesado de la información que llegue de la Raspberry Pi.

Una vez llegados a este punto, el programa entra en el bucle principal. Este bucle se ha construido mediante un *while error==Flase:* esto permite que el programa se ejecute hasta que la condición deje de cumplirse. Esto es importante porque si no se hubiera puesto esta condición el programa se ejecutaría continuamente y solo se podría parar mediante la desconexión de la alimentación de la placa Pyboard.

```
while error==False:
    mode=itera%2
    text='numero de iteracio:  ' +str(itera)+'  mode: '+str(mode)+'\n'

    #---
    # pampalluga LED
    if itera%50==0:
        blnk.toggle()
        text+='blinking\t ..... ok\n'

    #---
    # interrupccio del programa + escriure a la sd
    if bot()==True:
        print(text)
        f.write(text)
        break
```

Como se observa en el código anterior, se puede ver el comando para iniciar el bucle, y posteriormente se calcula el modo. El modo solo es el residuo de la división del número de iteración entre dos. Esto quiere decir que cuando la iteración sea un número par, el modo será *Cero*, y si la iteración es impar el modo será *Uno*. Esta diferencia en el modo indicará qué datos se envían por la UART. Esto se ha decidido así, ya que enviar cadenas de aceleración y giroscopio convierte a las tramas en extremadamente largas, lo que implica que la posibilidad de que aparezcan errores aumenta drásticamente. Por ello, se ha separado en dos y cuando el modo es *Cero* se enviarán los datos del acelerómetro, y cuando el modo es *Uno* se enviarán los datos del giroscopio. Posteriormente se escribe en la variable que guardará el reporte el número de iteración en la que se encuentra y el modo de ésta.

En el fragmento anterior también se observan dos pequeñas condiciones: la primera hace referencia al parpadeo del led número 2 (verde) para comprobar que la aplicación funciona según lo previsto. Se ha decidido que un buen valor son 50 iteraciones. Por ello, el estado del LED cambiará mediante el comando *blnk.toggle()* y se guardará en el reporte que el LED ha cambiado. La segunda condición hace referencia a la interrupción voluntaria de la aplicación. Una vez el código llega a este

punto se comprueba el estado del botón USR. Si este no está apretado, el resultado será *False* y no se ejecutará el código que tiene en su interior. Por otra parte, si el usuario decide apretarlo, al cumplirse la condición, el programa procederá a grabar en la tarjeta microSD todo lo que tenga en la variable *text* y ejecutará el comando *break*. Este comando lo que hará es salir automáticamente del bucle del programa principal dirigiéndose inmediatamente al final del código y última de las partes.

Pasada esta parte del código se entra en la parte que hace referencia a la comunicación entre dispositivos. Este fragmento es considerablemente breve en lo que respecta al código en el programa principal, pero como se explicará a continuación tiene mucho trabajo de programación.

```
text,dades=aux.mpuI2C(i2c,text)
text=aux.enviar(uart,mode,dades,text)
cua,text=aux.rebre(uart,'',text)
report=aux.procesa(cua)

text+='\t\t'+str(report[0])+' trames ok\t '+str(report[1])+' trames error\n'
totaltram[1]+=report[1]
totaltram[0]+=report[0]
#-----
text+='iteracio acabada\n'
itera+=1
f.write(text)
print(text)
```

Como se puede ver en el fragmento anterior, el cuerpo de la aplicación queda resumido en 4 líneas de código: primero se recopilan los datos del sensor I2C, posteriormente se envía la información por la UART y a continuación se lee todo lo que ha llegado y está almacenado en el “buffer”. Por último, se procesa la información que ha llegado. En el capítulo siguiente se explicará con más detalle todo lo que en realidad está sucediendo en el microcontrolador y está programado en los archivos en los que se apoya el programa principal.

Una vez se acaban todos los comandos que interactúan con los buses y procesan la información, quedan los comandos en que se usa la información recibida. En el caso de esta aplicación solo se guardan el total de tramas que la Raspberry Pi ha recibido correctamente y las que ha recibido con error. Posteriormente se procede a indicarle

al reporte que la iteración se ha acabado y guardarla en la tarjeta microSD. También se actualiza la iteración para que ésta vaya aumentando. Durante muchos de los fragmentos del texto se han podido ir viendo comandos *print* en algunas partes del código. Esto solo se usa cuando se trabaja en el modo emulador, para que por el terminal se vaya mostrando información. Este comando ralentiza la ejecución del código, por lo que en la aplicación definitiva se habrá retirado.

Por último está la parte que hace referencia al final de la aplicación. Cuando se ha decidido acabar el programa intencionadamente por el usuario, esta última parte solo sirve para grabar el total de tramas enviadas correctamente y las que han tenido errores, para posteriormente cerrar el fichero abierto para el reporte y apagar todos los LEDs para evitar el posible consumo de electricidad sin sentido.

```
f.write('trames enviades [ok/error] = '+ str(totaltram))
f.close()

pyb.LED(1).off()
pyb.LED(2).off()
pyb.LED(3).off()
pyb.LED(4).off()
```

3.3.2 Archivo funcions_auxiliars.py

Este archivo contiene las funciones que ayudan y complementan el archivo main.py para que se pueda ejecutar correctamente.

3.3.2.1 Función mpuI2C

Esta función es la encargada de preguntar al acelerómetro-giroscopio por la orientación en el espacio del dispositivo. Es una función que a su vez llama a otro archivo que es el que tiene los comandos para preguntar mediante el bus I2C toda la

información que necesita. Este fichero (*imu.py*), con la ayuda del fichero *vector3d.py,* es el encargado de recibir la información y convertirla en los valores que se van a usar. Este proceso es complicado y está resuelto por unos ingenieros del M.I.T y modificado para que funcione en este caso pero no se explicará con detalle (para más información del código se puede consultar el anexo).

```
def mpuI2C(bus,text):
    text+='\n'
    text+='\t----- '+'MPU6050 I2C-bus'+' -----\n'
    buff=(bus.accel.x,bus.accel.y,bus.accel.z,bus.gyro.x,bus.gyro.y,bus.gyro.z)
    text+='accelerometre:\t\tgiroscopi:\n'
    text+=str(buff[0])+'\t\t\t'+str(buff[3])+'\n'
    text+=str(buff[1])+'\t\t\t'+str(buff[4])+'\n'
    text+=str(buff[2])+'\t\t\t'+str(buff[5])+'\n'
    return text,buff
```

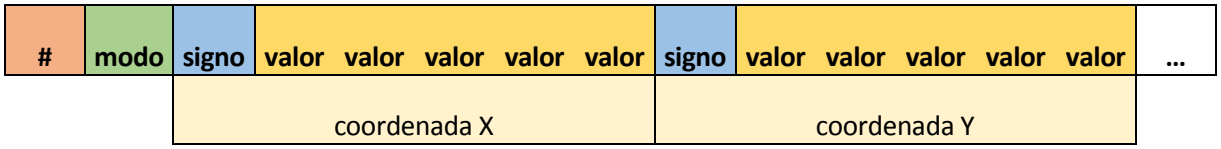
Esta función, como se ve en el fragmento superior, recibe dos elementos, el reporte para completarlo con las acciones que haga y el bus de datos por donde solicitar la información. Esta información, una vez procesada por otras funciones auxiliares se guarda en una tupla de 6 elementos: los tres primeros corresponden a los valores de la aceleración y los 3 últimos a los del giroscopio, siempre siguiendo el orden de las coordenadas (x, y, z). Por último se devuelven las dos variables, primero el reporte modificado y después los valores de posicionamiento de la IMU.

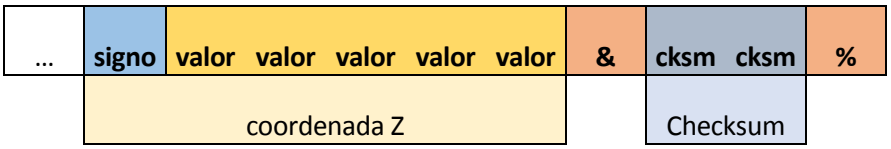
3.3.2.2 Función enviar

Esta función es probablemente la más importante de todo el proyecto, ya que es la encargada de que los datos se envíen correctamente. Para ello se reciben cuatro elementos: el bus por donde se tiene que enviar, el modo de envío (0 para aceleración, 1 para giroscopio), los datos a enviar y por último el reporte para ser completado.

```
def enviar(bus,mode,dades,text):
    text+='\t----- '+str(bus)+' -----\n'
    text+='   escrivint bus: \t'
    string=prepara(mode,dades)
    text+=string+'\n'
    bus.write(string)
    return text
```

Primero de todo, la función actualiza el reporte, indicando qué bus es y lo que va a escribir en el bus. Posteriormente llama a la función *prepara,* que será la encargada de convertir los datos que le han entrado a la función al formato que se ha decidido que sería adecuado para el envío. Este formato es el siguiente:





3.3.2.3 Función prepara

Como se puede observar, todas las tramas tienen la misma longitud. Cada trama se compone por diferentes elementos: el primero de todos es el carácter "#", que indica el inicio de una nueva trama, seguido de él va el modo de envío, "A" para

aceleraciones y “G” para los valores del giroscopio. Después de estos dos caracteres van los datos, siempre de igual longitud, constituidos por 6 caracteres. El primero es el signo del valor: “+” para positivos y “-“ para negativos, se escriben los signos para evitar confusiones en la lectura de valores y que las tramas tengan longitudes variables, ya que esto podría complicar la lectura, y posteriormente 5 dígitos que indicaran el módulo de la magnitud de la componente.

En el caso de las aceleraciones los cinco dígitos hacen referencia a un valor decimal de entre -9.9999 hasta +9.9999. Esto significa que el primer número es un entero y los cuatro que lo acompañan serán los decimales. En el caso del giroscopio es un poco diferente. Los signos funcionan del mismo modo y los números también tienen 5 cifras, pero en este caso, como los valores van de entre el -360 hasta el +360, se requieren 3 dígitos para la parte entera. Por ello solo tienen dos dígitos para la parte decimal. Este hecho no afecta demasiado ya que un pequeño cambio en los decimales no afecta mucho cuando se manejan valores del orden de las centenas.

Por último solo faltan 4 caracteres, que son los encargados del código de detección de errores. Una vez llegan todos los valores numéricos, para comunicar que se ha acabado de enviar los valores, se envía el carácter “&”. Este carácter indica que los dos siguientes valores corresponden al “checksum”. Este “checksum” es el encargado de verificar que el mensaje ha llegado correctamente. Para este cometido se usará la suma aritmética de todos los dígitos enviados, y se cogerá solo el residuo de dicha suma dividida entre 100. Esta operación proporciona solo dos valores (bytes). Se ha decidido que dos valores son suficientes ya que la probabilidad de que el “checksum” dé por bueno un mensaje erróneo es casi nula.

Por último se envía el carácter de fin de trama “%”. A continuación se puede ver el código de la función que prepara los datos para ser enviados:

```
def prepara(mode,dades):
    string='#'
    cksm=0
    if mode==0:
        string+='A'
        for i in dades[0:3]:
            if i>=0:
                string+='+'
            else:
                string+='-'
            string+=str(abs(int(i*10000)))
            for j in str(abs(int(i*10000))):
                cksm+=int(j)
    elif mode==1:
        string+='G'
        for i in dades[3:]:
            if i>=0:
                string+='+'
            else:
                string+='-'
                i=abs(i)
            num=(str(int(i/100))+str(int((i%100)/10))+str(int(i%10))+str(int(
abs((i-int(abs(i)))*100))))
            string+=num
            for j in num:
                cksm+=int(j)
    string+='&'+str(cksm%100)+'%'
    return string
```

Cuando la función *enviar* tiene la trama preparada, procede a actualizarla en el reporte y con el método *write(string)* de la clase UART proporcionada por MicroPython se escribe la información en el bus, que llegará al dispositivo XBee de radiofrecuencia.

3.3.2.4 Función rebre

Esta función es la encargada de recibir toda la información que llega por la UART y guardarla en una cola para su posterior procesado. Gracias al hardware que tienen estos pines especiales no hace falta que se habilite una interrupción cada vez que llegan datos, ya que el microcontrolador dispone de un *buffer* donde se almacenan

los datos que llegan de manera asíncrona.

Cuando se ejecuta la función *rebre* que la acompaña la información necesaria (el bus por donde se reciben los datos, la cola donde almacenar la información y el reporte que tiene que actualizar), ésta lo que hace es preguntar al microcontrolador si hay algun elemento en el buffer. En caso afirmativo, lo guarda en la variable *msg* (message), que es del tipo *bytes.* Este proceso iterativo acaba cuando se hayan leido todos los elementos del buffer. Cuando se ha acabado de leer se convierte la variable del tipo *bytes* al tipo *string*. Esto se realiza mediante el comando *str(msg).split(" ' ")*. Una vez convertido el mensaje, se guarda en la cola y se retorna la información y el reporte.

```
def rebre(bus,cua,text):
    text+='\t----- '+str(bus)+' -----\n'
    text+='   llegint bus: \t'
    msg=b''
    while bus.any():
        char=bus.read()
        msg+=char
    msg=str(msg).split("'")[1]
    text+=msg+'\n'
    cua+=msg
    return cua,text
```

3.3.2.5 Función procesa

Como se ha visto anteriormente en el fichero *main.py* y tal y como se ha comentado, la información que llega por la UART no está procesada y llega continuamente. Para ello, cuando se recibe se guarda toda en una cola. En este apartado, la función *procesa* es la encargada de transformar toda la información de la cola en datos que pueda interpretar el microcontrolador.

Primero de todo se tiene que conocer el formato de los datos que envía la Raspberry Pi. Este formato es conocido ya que el código que se ejecutará en el microprocesador

también ha sido escrito para el desarrollo de este proyecto. En este caso las tramas recibidas tendrán el siguiente formato:

numero 1	**X**	**X**	**X**	**numero2**	

Como se puede observar las tramas se componen de 4 elementos: el elemento numero 1 indica el número de tramas que se han recibido correctamente durante el último proceso de datos de la Raspberry Pi, el elemento numero 2 por el contrario son las tramas que no se han leído bien o contenían errores. También se han incorporado les elementos X (en verde). Estos sirven para separar los valores 1 y 2, ya que estos no son de longitud fija. Pueden tomar valores entre el 0 y el número de tramas que haya llegado ese momento. Por último está el último byte (naranja), que es el de final de trama. Este carácter servirá para poder diferenciar las diferentes tramas que lleguen al microcontrolador.

Toda la información almacenada en la cola se procesará para obtener los valores de mensajes recibidos correctamente e incorrectamente. Para ello, la cola se transformará en una lista mediante el comando *Split(),* donde cada elemento será una trama recibida. Posteriormente se evaluará cada trama por separado y se guardarán los valores de tramas correctas y de tramas incorrectas en la variable *llista*. Finalmente, como la cola quedará vacía, se retornará la lista con los valores deseados.

```
def procesa(cua):
    llista=[0,0]
    reports=cua.split()
    while len(reports)>0:
        elem=reports.pop(0)
        elem=elem.split('x')
        try:
            llista[0]+=int(elem[0])
            llista[1]+=int(elem[-1])
        except:
            pass
    return llista
```

4. Raspberry Pi

En este apartado, se explicará la función que desempeña la Raspberry Pi, junto con todo la explicación del código más importante (el resto se puede consultar en el anexo).

Como se ha comentado en la introducción al hardware, la Raspberry Pi es un microcomputador. Esto significa que es como un ordenador pero en pequeño. Al tener unas dimensiones similares a las de una tarjeta de crédito, éste no dispone de pantalla, ratón y teclados propios, pero puede llevar a cabo las mismas tareas que un ordenador convencional.

Para poder visualizar lo que se está haciendo en la Raspberry Pi, es necesario conectarle algún elemento. Para ello, dispone de un puerto de salida HDMI donde se le puede conectar una pantalla o proyector. Debido a que actualmente la mayoría de pantallas de ordenador aún funcionan con un conector VGA, se ha adquirido un adaptador VGA-HDMI. Gracias a este conector se podrá visualizar la Raspberry Pi en la pantalla sin ningún tipo de problemas. Además, se conectará también en los puertos USB que dispone un teclado y un ratón para interactuar con el dispositivo.

La función principal que desempeñará la Raspberry Pi es servir como punto donde se recopilan los datos que envía el microcontrolador y, posteriormente, procesarlos para que el usuario pueda tener una idea de lo que está sucediendo. En el futuro, sería una idea muy interesante implementar una manera que permita al usuario, desde la Raspberry Pi, interactuar con el microcontrolador y controlar algunos actuadores extras (por ejemplo servomotores).

4.1 Primer encendido

Primero de todo, ya que la Raspberry Pi es un computador, ésta necesitará un sistema

operativo. De todos los que se recomiendan en la página web oficial de Raspberry Pi, se ha elegido el sistema Raspbian.

“Raspbian es un sistema operativo libre basado en Debian y optimizado para el hardware Raspberry Pi. Un sistema operativo es el conjunto de programas básicos y utilidades que hacen que su funcionamiento Raspberry Pi. Sin embargo, Raspbian ofrece más que un SO puro; viene con más de 35.000 paquetes, software pre-compilado en un formato que hace más fácil la instalación en su Raspberry Pi. “ – página oficial de Raspbian.

Figura 10. Raspbian

El sistema operativo se debe instalar en una target microSD con una capacidad de más de 8GB. En este caso se usa una de 16 GB y conectada al ordenador mediante un adaptador USB. Para poder grabar la imagen del sistema operativo se ha descargado un programa llamado Win32DiskImager, debido a que hay muchos tutoriales en internet sobre cómo realizar esta tarea no se profundizará en este aspecto, pero el aspecto que tendría el programa seria el siguiente (véase Fig. 11).

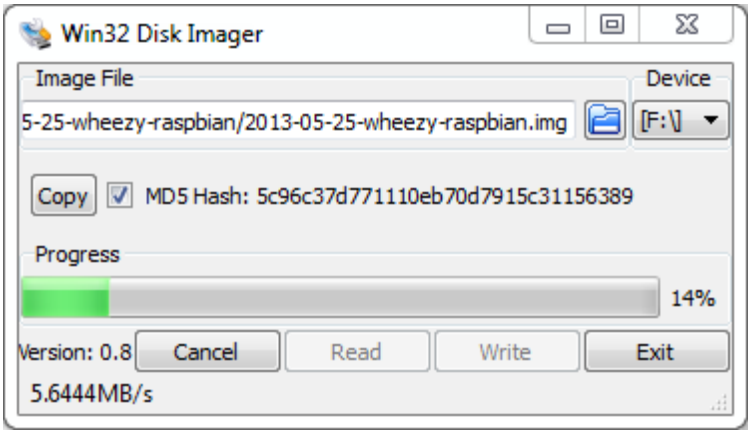


Figura 11. Captura del programa Win32 Disk Imager

Una vez grabada la imagen en la tarjeta microSD, insertada en la ranura y alimentado el computador, por la pantalla aparecerá un menú de configuración inicial (véase Fig. 12). Este menú inicial permitirá configurar diversas cosas, pero la más importante es, decirle al ordenador que deseamos que cuando se inicie se cargue el escritorio. Mediante la flechas del teclado, desplazarse hasta la opción*boot_behaviour* y cambiar a modo *desktop*. De esta manera la opción por defecto será cargar el escritorio. De lo contrario, no se dispondría de interfaz gráfica y al iniciarse se abriría el terminal. También hay opciones de configurar la zona horaria, el lugar y la opción de cambiar la contraseña y el usuario.

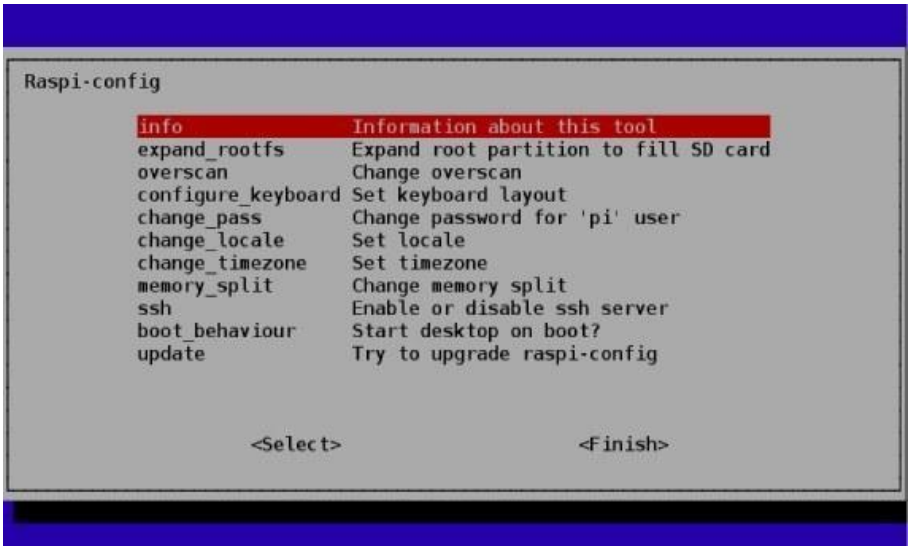


Figura 12. Menú configuración inicial de la Raspberry Pi

Cuando la configuración esté completa nos movemos hasta la opción que aparece debajo a la derecha de la pantalla (<finish>) y apretamos aceptar. La Raspberry Pi procederá a ejecutar los cambios y posteriormente se iniciará y aparecerá el escritorio.

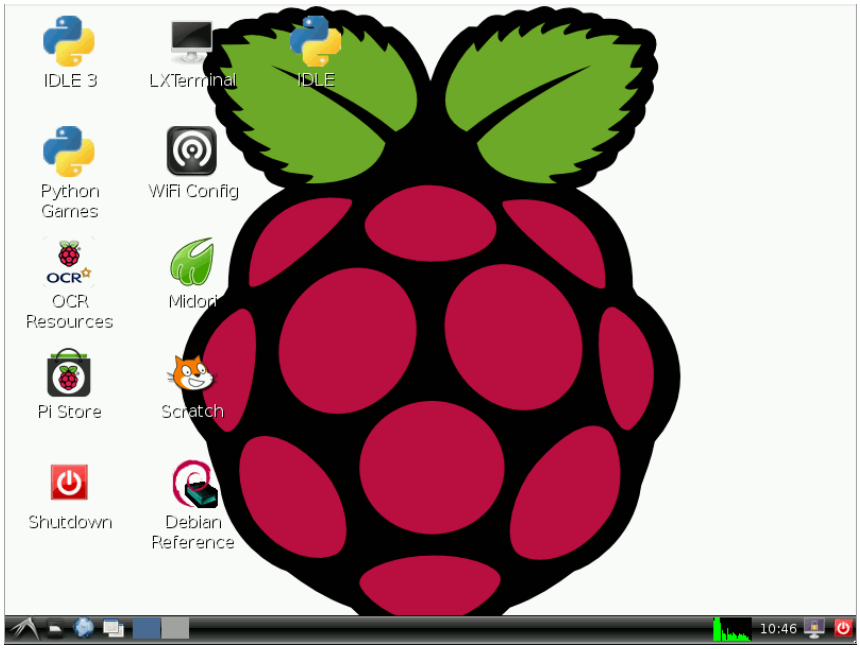


Figura 13. Escritorio de la Raspberry Pi

Como se puede ver en la figura superior (véase Fig. 13), el escritorio tiene un aspecto muy similar al que se puede ver en otro equipo con Debian, y es muy parecido al que tiene el dispositivo que usa este proyecto. Se recomienda que cuando se inicie por primera vez se ejecuten los siguientes comandos en el terminal: *sudo apt-get update* y *sudo apt-get upgrade*¸ y asentir a la instalación. Estos comandos son los encargados de actualizar la Raspberry Pi. Es conveniente que se ejecuten también de vez en cuando ya que es una plataforma que se mejora muy a menudo y se corrigen errores.

Una vez todo esté configurado y listo, se deberá conectar mediante el *XBee explorer* el módulo de radiofrecuencia con el computador. Los dispositivos de comunicarán mediante un cable USB-miniUSB. Gracias a que la Raspberry Pi B+ dispone de 4 puertos USB (los modelos A, A+ y B solo disponen de 2 puertos), se podrá conectar, además del ratón y teclado, el módulo XBee y una memoria Flash USB (pendrive) para cargar el código que se ejecutará. Así es como debería quedar el montaje final (véase Fig. 14)

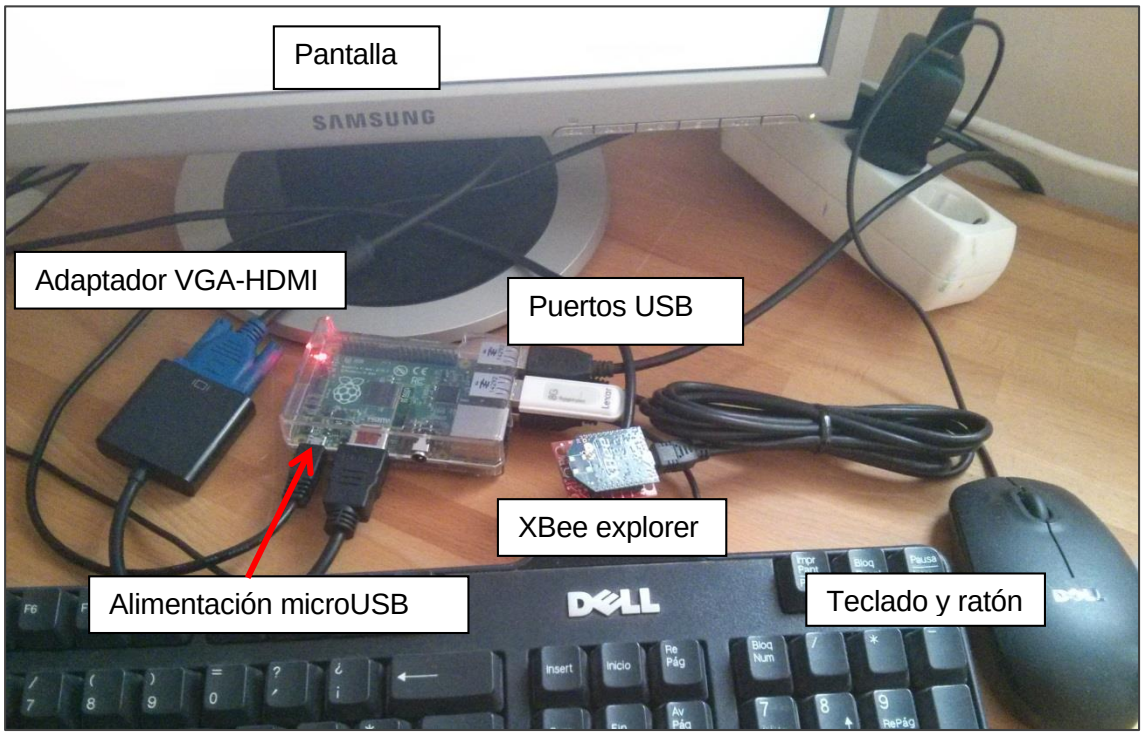


Figura 14. Montaje final de la Raspberry Pi

Cuando se tenga toda la Raspberry Pi configurada y operativa, se seguirá a crear el código necesario para recibir la información del microcontrolador.

4.2 Clase control

El objetivo de esta clase, que irá guardada en el fichero *control.py*, será el de gestionar toda la información que llegue a través del módulo XBee. Esta clase será la encargada de abrir el canal de comunicación, gracias al módulo pySerial, y posteriormente extraer la información de todas las tramas que vayan llegando.

Para ello, dentro del fichero se implementará el código necesario para crear una clase. Al inicializarse abrirá el puerto donde está conectado el XBee. Como se puede ver en el siguiente fragmento de código, que pertenece a la inicialización, hay el comando *try*. Este comando se utiliza porque, como no se sabe en qué puerto está el dispositivo, si este no está en el */dev/ttyUSB0,* entonces saltará la excepción e intentará abrir el */dev/ttyUSB1*. Esto se hace aunque normalmente se abre primero el puerto USB0, a veces puede que el computador lo tenga ocupado por algún motivo, entonces se abrirá en el USB1.

```
class control:
    def __init__(self):
        try:
            self.uart=ps.Serial("/dev/ttyUSB0",baudrate=9600, timeout=5)
        except:
            self.uart=ps.Serial("/dev/ttyUSB1",baudrate=9600, timeout=5)
        self.cua=''
        self.resultat=[0,0]
        self.dades=[0,0,0,0,0,0]
```

Como se ha podido ver, esta inicialización no es tan complicada como la de los objetos de la interfaz gráfica (se verán en el siguiente apartado). Esto es debido a que solo es necesario abrir el canal de comunicación y asignárselo a una variable. Después, cuando necesitemos algo de ese canal, solo lo tendremos que llamar, ya que estará abierto. Es un procedimiento similar al que se ha visto en el microcontrolador. Las funciones trabajan sobre los objetos que se le dan como parámetro y ejecutan o devuelven el resultado.

Posteriormente se creará el atributo *self.cua,* que será el lugar donde se guarde la información hasta que pueda ser procesada. También se crearán los atributos *self.resultat* y *self.dades,* que respectivamente permiten contar el total de tramas correctas e incorrectas que lleguen, y guardar el valor de los ejes del acelerómetro y giroscopio.

Una vez inicializada la clase, el proceso que la Raspberry Pi debe llevar a cabo es cíclico: debe leer lo que le llega por la UART, descodificarlo, generar una respuesta y por último enviar la respuesta. Por esta razón se ha implementado una función llamada *repeteix* que será la encargada de generar este bucle.

```
def repeteix(self):
    while True:
        ara=self.comunica()
        print ara
        time.sleep(0.02)
```

Como puede apreciarse, el fragmento anterior hace referencia a la función *repeteix*, donde el objetivo de la función es repetir ad eternum la función *self.comunica*. Esta función es en esencia un ciclo de los que se han descrito anteriormente, como se puede ver en el fragmento siguiente: el código lee, desentrama, descifra y envía el reporte generado en el ciclo. Para ello, usa las funciones *self.desxifra* y *self.desentrama* (se puede consultar el código en los apéndices).

```
def comunica(self):
    #self.dades=[accX,accY,accZ,roll,pitch,yaw]

    self.llegirUART()
    report=self.desxifra(self.desentrama())
    self.escriureUART(report)
    return self.dades


def llegirUART(self):
    while self.uart.inWaiting():
        char=self.uart.read()
        self.cua+=char

def escriureUART(self,data):
    self.uart.write(data)
```

La función *self.desentrama* es la encargada de separar todas las tramas que llegan, y son devueltas para su posterior proceso. Esta tarea se ve simplificada gracias a la forma que se le ha dado a los datos enviados por el microcontrolador, gracias a los caracteres de inicio de trama "#" y final de trama "%", la función solo tiene que buscar los caracteres e ir cortando fragmentos de la cola donde está la información que ha llegado. Si al final queda una trama a medias, como no se encuentra el carácter de fin de trama, la información se queda en la cola a la espera de ser completada la trama.

Por último, la función *self.desxifra* recibe las tramas completas que retorna la función

self.desentrama, comentada anteriormente. El proceso para obtener los datos es simple, analiza cada una de las tramas que recibe por separado, machacando la información redundante. Es decir, si llegan dos tramas que hacen referencia a los valores del acelerómetro, guardará los datos de la última trama que ha llegado correctamente. Primero de todo se verifica que la trama cumpla el *checksum.* Si no lo cumple salta una excepción que automáticamente deja la trama como errónea y la descarta. Después de comprobar el *checksum* separa los dos posibles modos y los signos de cada componente y, por último, guarda los valores en la variable *self.dades*.

Después de analizar todas las tramas que ha recibido la función, genera un reporte del número de tramas que han llegado correctamente y las que no. Esta información se envía por la UART al microcontrolador y se almacena en la variable interna *self.resultat* para que, una vez acabado el proceso, tener un cómputo total de tramas que ha llegado correctamente y erróneas para realizar un pequeño estudio de prestaciones y alcance efectivo.

5. Interfaz gráfica

Durante el desarrollo de esta aplicación se decidió incluir una pequeña y muy simple interfaz gráfica. Ésta permitirá al usuario conocer la información que envía por radio frecuencia el microcontrolador y disponer de una interpretación simple, rápida y muy visual. Para ello, se ha utilizado una herramienta de programación llamada Pygame.

Pygame es un conjunto de módulos que permite y facilita el desarrollo de juegos en lenguaje Python. En este caso no se utilizará para crear un juego, ya que la interacción no será virtual debido a que recibiremos información en tiempo real de lo que sucede en nuestro microcontrolador. En esta aplicación se usará para mostrar por la pantalla el resultado de procesar la información recibida por nuestro microcomputador por radio frecuencia.

La estructura del programa principal es muy similar a la que usan los estudiantes de informática 2 en la ETSEIB para el desarrollo de su juego. El esquema en el que se ha basado esta aplicación consiste en: primero se inicializan las clases que formaran la interfaz gráfica y la clase que controlará la comunicación RF y posteriormente se define la pantalla con sus dimensiones y el reloj. Cuando todo está inicializado, se entra en un bucle infinito que solo se detendrá cuando se reciba la orden de salir (pygame.QUIT). En este bucle se ejecutara periódicamente la clase control, que es la encargada del correcto funcionamiento de la aplicación, y proporcionará la información necesaria para poder actualizar todas las otras clases que mostrarán la información.

```
pygame.init()
inicialitzacions de les classes
pantalla = pygame.display.set_mode(midesPantalla)
classes = pygame.sprite.OrderedUpdates() # grup de Sprites que
                                         # s'actualitzaran i es dibuixaran
                                         # seguint l'ordre d'inserció.
afegim els sprites que volem agrupar mitjançant classes.add()
crono = pygame.time.Clock()
final = False
while not final:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            final = True
        else examinem altres tipus per fer el que estigui previst

    actualitzar control i les classes...
    classes.update()        # Actualitza l'estat de cada sprite de la llista
    pantalla.fill(colorFons)# Pinta el fons de la pantalla
    classes.draw(pantalla)  # Pinta tots els sprites de follets
    pygame.display.flip()   # visualitza la nova pantalla
    crono.tick(24)          # limita a 24 imatges per segon (24 fps)
pygame.quit()
```

En la fase del diseño de la interfaz gráfica se ha tomado la decisión de cómo presentarle al usuario la información recopilada por el acelerómetro y giroscopio conectados al microcontrolador. Para ello se usarán 3 gráficos de barras para representar la aceleración (una para cada eje) y basándose en los sistemas de aeronavegación se ha representado también un horizonte artificial y una brújula.

El horizonte artificial es el instrumento básico con el que los pilotos determinan la

actitud espacial de la aeronave. Muestra la orientación longitudinal de la aeronave (la relación del eje longitudinal del avión con respecto al plano del suelo), es decir: si está inclinado (alabeo o roll, alas inclinadas hacia un lado u otro), con el morro levantado o bajado (cabeceo o Pitch, morro arriba o abajo del horizonte) o todo a la vez.

Sirve de gran ayuda en condiciones en que la visibilidad es poca o nula. El horizonte artificial tiene dos partes: el horizonte propiamente dicho, y el indicador de rumbo. El primero está compuesto por una región azul que representa el cielo, otra normalmente marrón que representa la superficie terrestre, una mira que representa el morro del avión, y varias marcas a su alrededor. Las marcas horizontales a ambos lados representan las alas, el plano de la aeronave, y su ángulo con el límite entre las regiones de cielo y superficie (el horizonte artificial), el ángulo de alabeo. Dispuestas verticalmente a intervalos regulares, hay marcas horizontales más pequeñas que representan ángulos concretos en el plano vertical, a intervalos de 5º, 10º, etc. Muestran el ángulo actual del eje longitudinal con el plano del suelo. Su principio mecánico de funcionamiento es giroscópico (véase Fig 15).

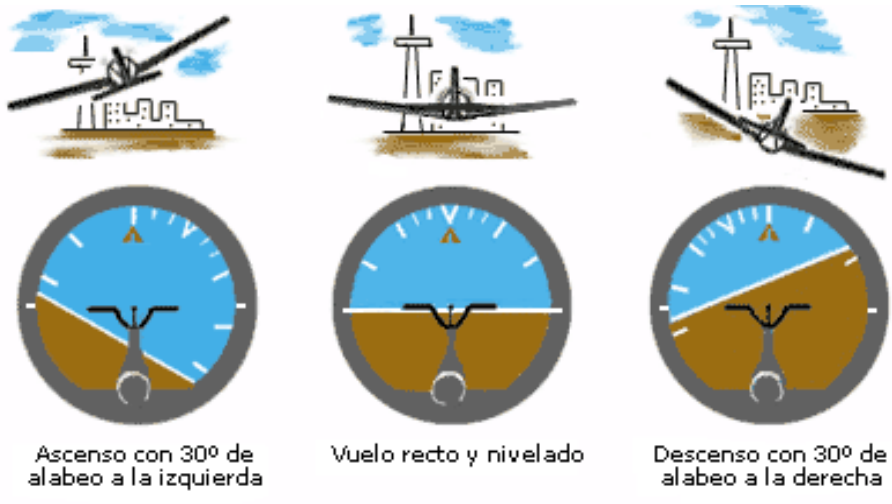


Figura 15. Ejemplo horizonte artificial

Por último también se ha representado un sistema que muestra la orientación del eje Z (yaw) de la IMU. A dicho elemento se le ha puesto el nombre de brújula, aunque en esta aplicación no es exactamente una brújula magnética, solo aportará el giro del

giroscopio respecto a la posición inicial.

Debido a que la memoria RAM de la Raspberry Pi es limitada, el funcionamiento de la aplicación se ve enormemente afectado, experimentando una ralentización del proceso que hace que sea inviable el control con la Raspberry Pi. Por este motivo se ha decidido que la interfaz gráfica se ejecutará en un ordenador (sin modificar el código) que dispone de más prestaciones y por la Raspberry Pi solo se mostrarán los datos recibidos por el terminal.

5.1 Acelerómetro

Esta clase permite crear un objeto con los métodos necesarios para la representación del valor de la aceleración en uno de los ejes. Esta clase crea una *Surface* con la forma y el fondo definidos donde se dibujarán las barras posteriormente. Estas barras reflejarán la magnitud de la aceleración en el eje correspondiente, con valores comprendidos entre -4g y +4g.

Como se puede ver en la figura siguiente (véase Fig. 16), es el aspecto que tendría la representación de la aceleración en el eje X. Los colores son diferentes en las diferentes barras para que la interfaz sea más visual.

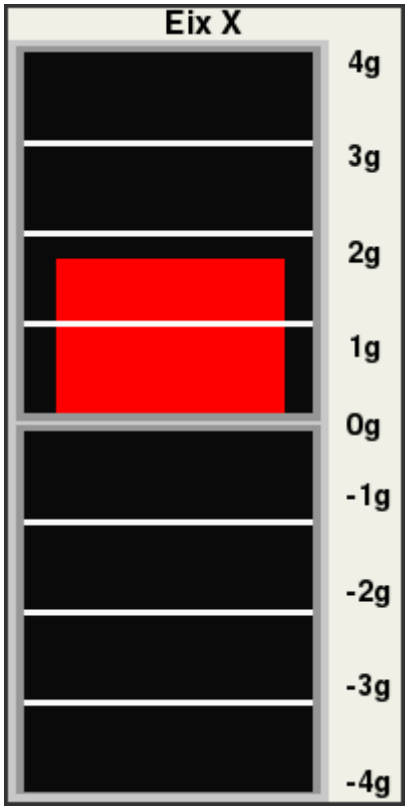


Figura 16. Gráfico de barras del acelerómetro

Para el diseño de la figura anterior se han utilizado diferentes funciones dentro de la clase para facilitar el dibujo. Primero de todo con la inicialización, se puede ver que se llama a diferentes funciones para crear el fondo o la base, y se guarda en el atributo *self.image*. El atributo *self.rect* también será muy importante, ya que indicará el lugar donde se tiene que poner la imagen en la ventana que abrirá la aplicación.

```
def __init__(self,pos,rec,text,color,colorbarra):
    pygame.sprite.Sprite.__init__(self)
    pygame.font.init()
    self.rec=rec
    self.espaillegenda=rec[0]*0.2
    self.espaititol=rec[1]*0.05
    self.dim=(self.rec[0]-self.espaillegenda,self.rec[1]-self.espaititol)
    self.marge=self.dim[0]*0.05
    self.text=text
    self.valor=0
    self.color=color
    self.colorbarra=colorbarra

    self.fondo=self.creafondo()
    self.image=self.creafondo()
    self.rect=self.image.get_rect(left=pos[0],top=pos[1])
```

Como se puede ver en el fragmento de código anterior (inicialización de la clase), solo se declaran y guardan las variables que resultarán importantes, y finalmente se crean los atributos *self.image* y *self.rect*, que como se han comentado anteriormente son los más importantes. El resto del código está disponible en el apéndice, con los demás archivos, todos completos. Llegados a este punto, es conveniente comentar las funciones *actualitza()* y *update()*, ya que son características de las interfaces de Pygame. Como se puede ver en el fragmento siguiente, son las encargadas de actualizar la información y crear el dibujo que se mostrará. Estas funciones son extremadamente importantes, ya que permiten que la interfaz cambie y muestre la información actualizada.

```
    def actualitza(self,valor):
        self.valor=valor

    def update(self):
        valor=self.valor
        cuadrat=self.crearecuadre()
        barres=self.barres(valor)
        cuadrat.blit(barres[0],(self.marge,self.marge-2))
        cuadrat.blit(barres[1],(self.marge,self.marge*1.9+barres[0].get_size(
)[1]))

        fondo=self.image
        fondo.blit(cuadrat,self.poscuadrat)
        self.image=fondo
```

La función *actualitza* es la encargada de cambiar el valor que tiene la clase acelerómetro. De esta forma el valor que mostrará la barra irá cambiando a medida que llegue nueva información. Para ver el código completo del archivo, véase el apéndice.

5.2 Giroscopio

La clase giroscopio es la encargada de crear una *Surface* en Pygame donde se mostrará la información referente a los ángulos de Pitch y Roll. Gracias a este elemento, junto con la brújula, se podrá conocer la orientación de la unidad de movimiento inercial (IMU). Este elemento puede ser muy útil, ya que si acoplamos la IMU a una herramienta u otro objeto, se podrá conocer su orientación. Esta información es de gran ayuda en la mayoría de aplicaciones de tele-operación, ya que ayuda al usuario a crear una imagen mental de la situación.

En la interfaz creada en este proyecto, el giroscopio tiene un aspecto como el que se puede apreciar en la siguiente figura (Fig. 17):

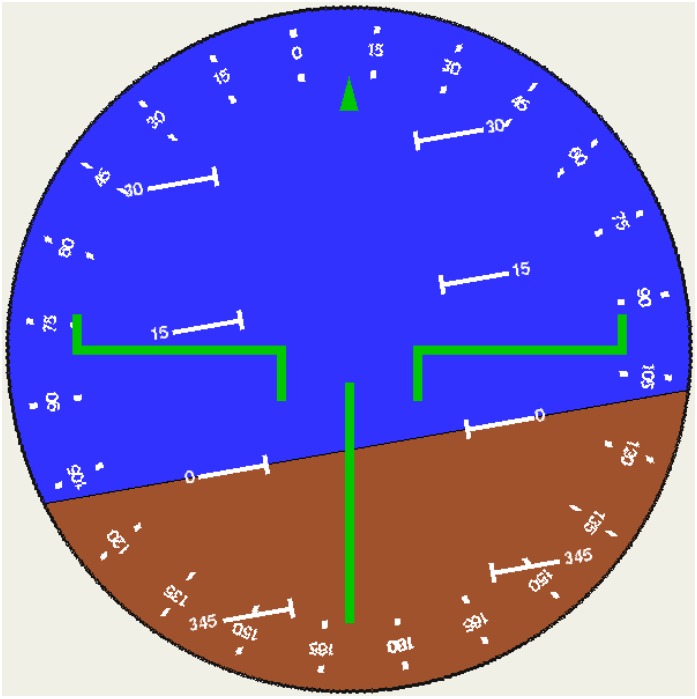


Figura 17. Gráfico del Horizonte artificial

Como se puede ver en la figura anterior, el aspecto recuerda al de un horizonte artificial igual que el que se utilizan en aviación. Para la construcción de este objeto, se ha requerido un procedimiento similar al que se ha comentado en el acelerómetro, pero con algunas modificaciones. Debido a que la forma no es la misma, el código es diferente pero sigue los mismos patrones: se puede comprobar en el fragmento siguiente, que hace referencia a la inicialización.

```
def __init__(self,pos,dim,color):
    pygame.sprite.Sprite.__init__(self)
    pygame.font.init()
    self.transparent=(255,0,255)

    self.roll=0
    self.pitch=0
    self.dim=dim
    self.pos=pos
    self.color=color
    self.radi=(0.5*self.dim)-5
    self.gruix=self.dim/150

    self.T=self.marcador()
    self.fondo=self.creafondo()
    self.sky=self.creasky()
    self.lletra=pygame.font.Font('freesansbold.ttf',20)

    self.image=pygame.Surface((self.dim,self.dim))
    self.rect=self.image.get_rect(left=pos[0],top=pos[1])
```

En este caso, la imagen del giroscopio necesita de más elementos para mostrar. Como se puede ver, se han escrito una serie de funciones (marcador, creafondo, creasky) que ayudan a crear la imagen final que se mostrará (todo el código adicional está disponible en el apéndice). Por último, también se puede ver cómo se declaran las variables *self.image* y *self.rect*, encargadas de guardar la información para cuando se tenga que actualizar la ventana principal.

```
def actualitza(self,roll,pitch):
    self.roll=roll
    self.pitch+=pitch
    if self.pitch>325:
        desfase=325-self.pitch
        self.pitch=-35-desfase
    if self.pitch<-325:
        desfase=-325-self.pitch
        self.pitch=35-desfase
```

Como se puede ver en el fragmento anterior, que hace referencia a la función *actualitza* de la clase giroscopio, el código no es tan sencillo como en el caso del acelerómetro, ya que la naturaleza del horizonte artificial es muy distinta a un gráfico de barras. Debido a que la unidad de movimiento inercial (IMU) puede rotar respecto un eje indefinidamente, es necesario procesar la información que llega. Como se puede ver en el ejemplo, se tienen que acotar los valores del ángulo Pitch entre -325 y 325. Estos valores aseguran que en la imagen de cielo artificial que se ha creado, siempre estará dentro de los límites.

En el caso de la función *update*, ésta es más compleja y más extensa que la del acelerómetro. Para obtener el resultado necesario se han tenido que transformar imágenes. Ya que esta imagen indica dos ángulos diferentes, el proceso para girar las imágenes es más extenso. Por ello no se mostrará en esta memoria, pero está disponible en el anexo.

5.3 Brújula

La clase *compas* es la encargada de crear un objeto que muestre en la interfaz gráfica el valor de la componente Z del giroscopio. En el caso de la brújula, la información que se muestra no es de gran complejidad. Por este motivo no se ha reparado mucho en la estética (véase Fig. 18). Este objeto está compuesto por una rueda negra con ángulos que van desde el -180 al +180 y un indicador triangular de color verde.

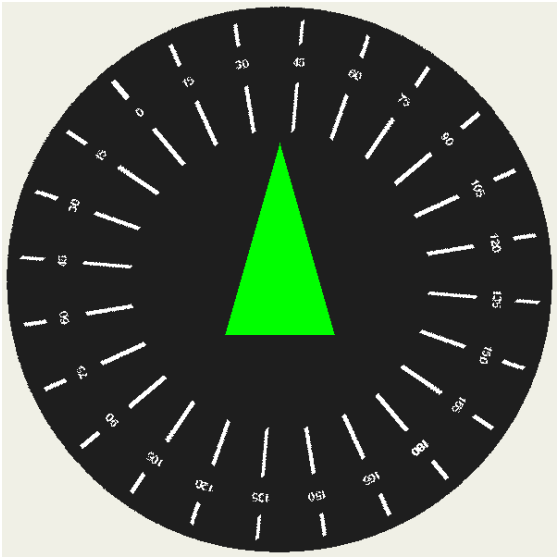


Figura 18. Gráfico de la brújula

Como las dos clases comentadas anteriormente, la clase *compas* tiene la misma estructura que las anteriores. Como se puede ver en la inicialización, los atributos *self.image* y *self.rect* son los más importantes, ya que indican qué imagen y dónde se tiene que colocar en la ventana. Como se puede ver en el código proporcionado a continuación, la estructura de la inicialización es extremadamente similar a la que se ha visto en el giroscopio. Esto se debe a que. el principio es el mismo pero en este caso solo se tiene que mostrar la información de un ángulo (eje Z o yaw).

```
def __init__(self,pos,dim,colorfons):
    pygame.sprite.Sprite.__init__(self)
    pygame.font.init()

    self.dim=dim
    self.yaw=0
    self.colorfons=colorfons
    self.radi=(0.5*self.dim)-5
    self.gruix=self.dim/150
    self.transparent=(255,0,255)

    self.T=self.marcador()
    self.fondo=self.creafondo()
    self.image=self.creafondo()
    self.rect=self.image.get_rect(left=pos[0],top=pos[1])
```

En el caso de la brújula, las funciones *actualitza* y *update* son similares a las del giroscopio. En la primera se actualiza la información del ángulo gracias a las tramas que llegan del microcontrolador, y en la segunda, se modifica la imagen que posteriormente será mostrada en la ventana principal. Gracias a la función *pygame.transform.rotate(image,angle)* se puede girar el disco con los ángulos, pero posteriormente se tiene que reajustar la imagen, ya que crea una imagen con un tamaño diferente. Una vez se tiene la imagen reajustada solo se debe pegar el marcador encima de ésta para tener la imagen definitiva.

```
    def actualitza(self,valor):
        self.yaw=valor

    def update(self):
        yaw=self.yaw
        nova=self.fondo
        nova=pygame.transform.rotate(nova,yaw)
        marge=(nova.get_size()[0]-self.dim)/2

        nova1=pygame.Surface((self.dim,self.dim))
        nova1.set_colorkey(self.transparent)
        nova1.blit(nova,(0,0),(marge,marge,marge+self.dim,marge+self.dim))

        posT=((self.dim-2*self.radi)/2,(self.dim-2*self.radi)/2)
        nova1.blit(self.T,posT)
        cercle=pygame.Surface((self.dim,self.dim))
        cercle.fill(self.colorfons)
        cercle.set_colorkey(self.transparent)
        pygame.draw.circle(cercle,self.transparent,(self.dim/2,self.dim/2),in
t(self.radi),0)
        nova1.blit(cercle,(0,0))

        self.image=nova1
```

6. Validación y estudio de prestaciones

En este capítulo se abordarán las cuestiones relacionadas con la puesta a punto de la aplicación y las características que tendrá a la hora de usarla. También se efectuará un pequeño estudio para ver las limitaciones de la comunicación por radiofrecuencia a diferentes distancias y a diferentes velocidades. En este punto también se tiene que tener en cuenta que el diseño es un prototipo. Por este motivo no se puede considerar que esté en el máximo de sus prestaciones, ya que en el futuro podría incluir mejoras y/o expansión en prestaciones e implementarse en un sistema con capacidad de ejecutar diferentes órdenes, no como ahora que es simplemente una forma de comunicación entre dispositivos.

6.1 Montaje final y hardware adicional

Este apartado se centrará en describir y comentar todo el material adicional que ha sido necesario para el montaje. Ya que se planteó que el microcontrolador fuera un sistema independiente, necesita de una alimentación y un regulador de tensión para no dañar los circuitos. Este regulador de tensión es un dispositivo de apenas 2x2 cm^2 (véase Fig. 19) que se encarga de que la tensión de salida sea constante y próxima a los 5V (5V Step-Up/Step-Down Voltage Regulator). Para alimentar el montaje se ha usado un porta-baterías de cuatro huecos para pilas AA 1.2V y 2200mAh. Con este montaje aseguramos que el microcontrolador siempre esté bien alimentado y sin sobretensiones (véase Fig. 20).

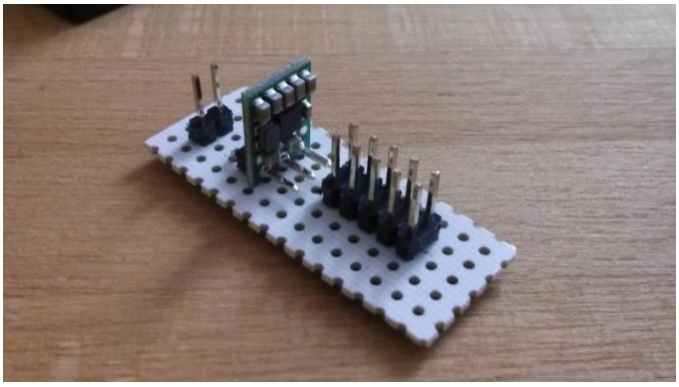


Figura 19. Regulador de tensión soldado en una placa de baquelita con pines para conexión

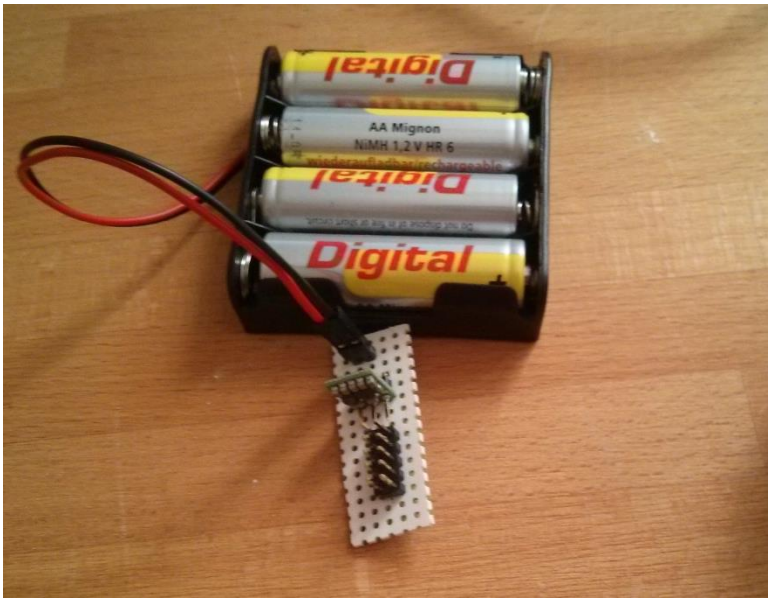


Figura 20. Regulador de tensión con baterías AA

En el caso de la Raspberry Pi se alimentará mediante un cargador de móvil de 5V y 2000mA. Esta característica (alimentación de la Raspberry Pi mediante microUSB) es de gran utilidad ya que hoy en día es muy fácil encontrar este tipo de conectores y adaptadores. Como se puede ver, el resultado final es parecido al que se puede ver en la siguiente imagen. Se puede ver la Raspberry Pi a la derecha, que mediante un adaptador VGA-HDMI se conecta a un monitor y por USB a un teclado y ratón.

También se puede ver que la conexión al módulo XBEE es mediante USB. Esto es debido a que como se usará el mismo programa para la conexión con la Raspberry Pi y con el ordenador, este último no tiene pines para la conexión.

Para tener una idea general de cómo sería aproximadamente el montaje final, se puede consultar la Figura 21. El microcontrolador, dotado con una batería, se puede mover fácilmente y responde a los objetivos planteados: ser un sistema autónomo con su propia alimentación. Una futura ampliación de este proyecto podría ser ensamblar todos los componentes en una estructura para que sean más portables y compactos.

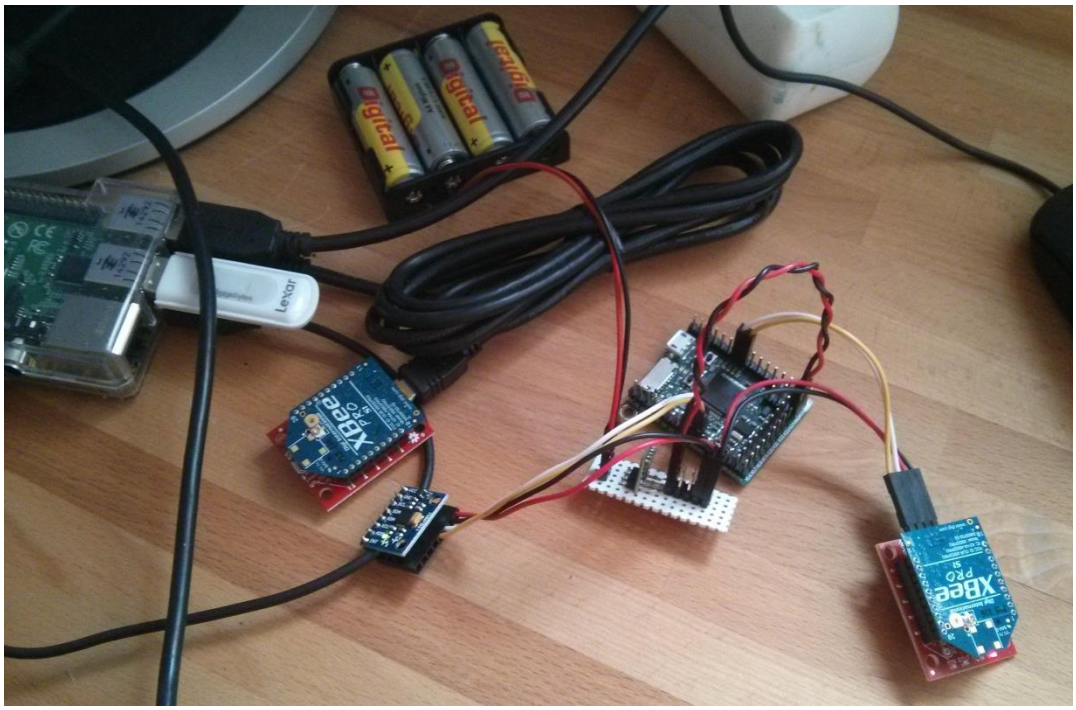


Figura 21. Montaje final

6.2 Validación de la aplicación

A causa de la complejidad de esta aplicación, lograr una comunicación estable no ha sido fácil. En capítulos anteriores solo se ha explicado el resultado del código pero no las alternativas que se plantearon y como se llegó a la solución final.

Para lograr la comunicación por UART en la Raspberry Pi previamente se tuvo que instalar el módulo *pySerial*. Este módulo permite usar los puertos de entrada y salida del computador en las fases iniciales de la aplicación, cuando se estaba probando que los dispositivos pudieran recibir y enviar datos. Para esta primera comunicación se usaba el modo emulador en el microcontrolador, y posteriormente a la inicialización del bus se escribía alguna información en él. El objetivo de esta prueba era asegurar que los dispositivos eran capaces de comunicarse manualmente antes de pasar las iteraciones. Durante este proceso también se inicializaba la UART en la Raspberry Pi y se le ordenaba que leyera todo lo que recibía.

Estas pruebas iniciales son extremadamente importantes, ya que permite averiguar el formato de datos que los dispositivos leen y escriben en el bus. Posteriormente se intentó crear las tramas que enviaba el microcontrolador de una forma distinta a la actual. En la versión actual las tramas se componen de una cadena de *strings*, pero en las primeras pruebas se intentó hacer de otra manera. La primera idea fue convertir todos los datos a bytes: el primer byte indicaría el comienzo de la trama y el modo, por ejemplo, el byte 0x5A era el encargado de iniciar la trama para el modo aceleración y el byte 0x5B para el modo giroscopio, de la misma manera el byte 0x5C sería el encargado de finalizar la trama. En el caso de los números se intentó convertirlos al formato estándar IEEE754. Esta conversión permitía transformar números decimales positivos y negativos en una cadena de 8 bytes. Estos números posteriormente se le invertiría la codificación para obtener el valor inicial (con un muy pequeño error). De esta forma las tramas también habrían sido de longitud fija (27 bytes en total, inicio de trama + 8 bytes x3 de datos + ckecksum + fin de trama).

Esta forma de enviar los datos no pudo implementarse finalmente, ya que había problemas en la codificación y descodificación de los datos, debido a que el microcontrolador funcionaba con una versión de python3 y el módulo *pySerial* de la Raspberry Pi solo soportaba hasta la versión Python2.7. Por este motivo se tuvo que adoptar la solución actual de cadenas de *strings*.

Por otra parte, también es posible escoger la configuración de la UART de los dispositivos. De entre los muchos parámetros que se pueden modificar, el que afectará más a la aplicación es el *baudrate* (bits por segundo del bus). Para las pruebas iniciales se ajustó a un valor conservador de 9600 bits/s para asegurar el buen funcionamiento, pero posteriormente se decidió modificar a 115200 bits/s. Este nuevo valor proporcionaría una velocidad mayor en el tráfico de datos, pero también aumentaría la probabilidad de error en las tramas. Por este motivo se ha realizado un estudio para ver cómo afecta la distancia entre los dispositivos y la velocidad de transmisión de datos a la fiabilidad de la comunicación.

6.3 Estudio de prestaciones

Como se ha comentado anteriormente, la fiabilidad de la comunicación depende de la distancia entre los dispositivos y la velocidad de transmisión. Por este motivo se realiza el siguiente estudio.

Para la realización del estudio hay que tener en cuenta que los módulos de radiofrecuencia XBEE tienen un alcance máximo de 1.5 km, pero en este caso, al no disponer de antena el alcance es mucho menor. Por este motivo se ha probado el alcance máximo en estas condiciones y se ha encontrado que el alcance máximo en exteriores es de 11 metros y en interiores es poco superior a los 6 metros.

Se ha propuesto el siguiente experimento: calcular el número de tramas que recibe correctamente la Raspberry Pi a diferentes distancias y a diferentes *baudrates*. Se contabilizarán las tramas enviadas en interiores, ya que es el caso donde se podría tener mayores problemas en la comunicación debido a paredes y otros obstáculos. Como se puede ver en la siguiente tabla, un recuento delos diferentes casos.

		baudrate							
		9.600 bits/s				115.200 bits/s			
		totales	ok	error	% ok	totales	ok	error	% ok
distancia	0,1 m	1027	1027	0	100,00	1355	1354	1	99,93
	0,5m	1249	1247	2	99,84	1139	1137	2	99,82
	1m	864	862	2	99,77	1075	1071	4	99,63
	2m	1428	1423	5	99,65	1384	1368	16	98,84
	3m	989	927	62	93,73	1002	916	86	91,42
	4m	831	613	218	73,77	1096	712	384	64,96
	5m	1106	432	674	39,06	907	195	712	21,50
	6m	1050	171	879	16,29	1207	83	1124	6,88

Tabla 1. Estudio sobre la eficiencia de la comunicación por Radio Frecuencia

Como se puede observar en la tabla anterior, se puede ver claramente una correlación entre la distancia a la que están los dispositivos y el número de tramas que llegan erróneamente. En el grafico siguiente (Fig. 22), también se puede observar que si se aumenta la velocidad de transmisión también se incrementa la posibilidad de que lleguen tramas incorrectas.

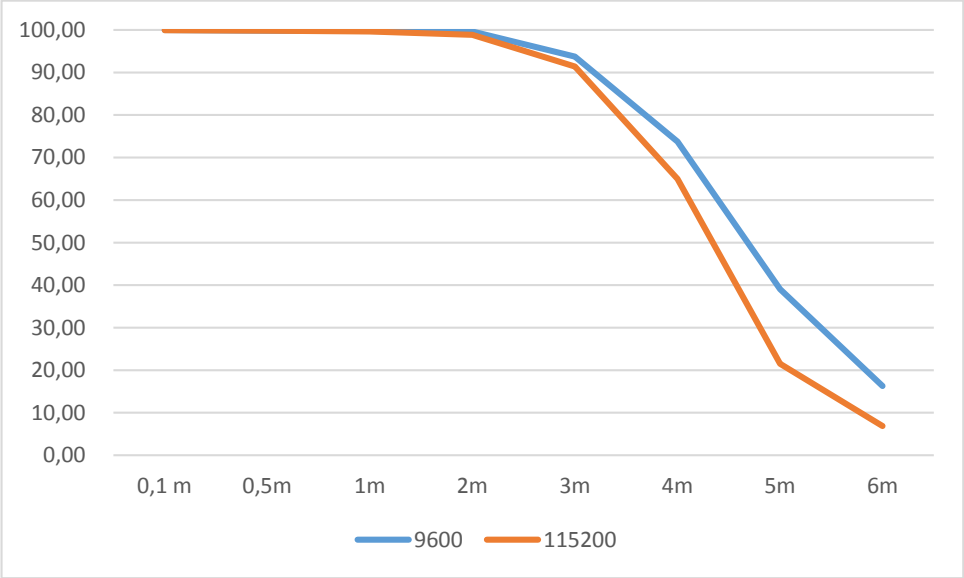


Figura 22. Gráfico de % tramas correctas en relación distancia de separación

Como conclusión de este pequeño estudio de prestaciones, se puede observar que dependiendo de la finalidad de la aplicación es mejor cambia la velocidad de transferencia de datos. Por lo general a distancias cortas es indiferente la velocidad, ya que el porcentaje de tramas que llegan correctamente es muy similar, pero se recomendaría utilizar la velocidad de 115200 baudios, ya que se podría enviar más información en el mismo tiempo. Por otra parte, cuando la distancia aumenta, se tiene que escoger qué cualidad es mejor, la fiabilidad de los datos o bien la cantidad de datos. También se debe comentar que para aumentar el rango de la comunicación se puede conectar una antena, ya que el chip dispone de un conector U-FL.

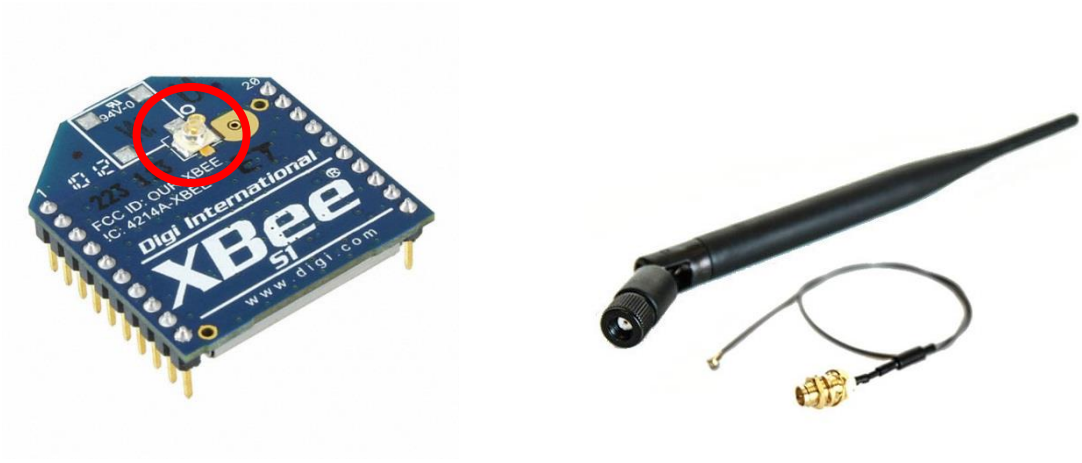


Figura 23. XBee con posible antena U-FL

Mediante una antena como la mostrada en la figura anterior (Fig. 23) se puede llegar a la distancia de 1.5 kilómetros aproximadamente, valor que nos proporciona las especificaciones del módulo XBee, pero ya que el objetivo no es lograr un gran rango de comunicación, no se han estudiado a fondo estas alternativas.

7. Presupuesto económico

Durante el desarrollo de cualquier proyecto se generan unos costes y gastos sujetos al material necesario para llevar a cabo dicho proyecto o bien el coste de tener personas trabajando en él. En este proyecto concretamente se ha tenido que adquirir el hardware necesario para la puesta en práctica de la comunicación por RF y también los dispositivos a comunicar (Raspberry Pi y la Pyboard con MicroPython). Por otro lado, el coste del software necesario para la configuración de los módulos XBEE (X-CTU) y para la creación y edición del código es nula, ya que son de código abierto y los proporciona la empresa creadora de los módulos de radio frecuencia.

Para la adquisición del hardware se ha decidido que el mejor método era la compra por internet, ya que permite un mejor contraste de los precios, ahorra el esfuerzo de tener que ir a una tienda física a comprar los productos (a veces a más de una) y se puede ir directamente al fabricante sin pasar por intermediarios. Por otra parte, también existen inconvenientes: el tiempo de espera para que llegue el envío puede llegar a superar las 2 semanas y siempre hay el riesgo de que se extravíe por el camino. En este caso se ha decidido comprar en las siguientes páginas web:

- MicroPython: se ha adquirido en la página oficial: https://MicroPython.org/store/#/store
- Raspberry Pi B+: se ha adquirido en la página web de Amazon: http://www.amazon.es/
- Módulos XBEE : se han adquirido en la página web Sparkfun Electronics: https://www.sparkfun.com/
- El hardware necesario para la conexión de los módulos RF se ha adquirido en la página web Sparkfun Electronics: https://www.sparkfun.com/
- Cables, conectores, crimpadores, batería, regulador de voltaje …se han adquirido en la página web Pololu: https://www.pololu.com/

Resumen detallado del coste del material:

Material			
Artículo	coste unidad (€)	nº unidades	coste total (€)
Raspberry Pi B+	33,5	1	33,5
MicroPython	35,75	1	35,75
XBEE	37,95	2	75,9
XBee explorer regulated	19,9	1	19,9
XBee explorer USB	24,95	1	24,95
MPU6050	2,15	1	2,15
Cable 26 AWG	4,5	4	18
Pines de conexión macho	7,95	1	7,95
Pines de conexión hembra	5,95	1	5,95
Conectores	0,99	1	0,99
Batería NiMH AA 1.2V 2200mAh	2,39	4	9,56
Portador baterías 4-AA	1,19	1	1,19
Regulador de voltaje 5V	4,95	1	4,95
		total =	240,74 €

Tabla 2. Resumen de los materiales con sus costes

Por otra parte, también se debe tener en cuenta el tiempo dedicado por los desarrolladores en el cómputo total para tener una idea aproximada del coste total de la realización del proyecto si fuera realizado en un ámbito no académico. Para este apartado se tendrá en cuenta de que solo ha trabajado una persona (un ingeniero/a) con una remuneración de 45€/hora.

Tiempo de trabajo			
	horas	coste (€/h)	coste total(€)
Estudio previo e investigación	50	45	2.250
Desarrollo del software	250	45	11.250
Prueba y validación	50	45	2.250
		total =	15.750

Tabla 3. Resumen de costes de trabajo

Como se puede ver en la siguiente tabla, el coste aproximado del proyecto serían poco menos de 16.000€.

Coste total del proyecto	
Material	240,74
Tiempo de trabajo	15750
total	15.990,74 €

Tabla 4. Resumen total de costes

9. Impacto medioambiental

Debido a la naturaleza de la aplicación desarrollada, se tiene que tener en cuenta dos tipos de repercusión sobre el medio ambiente. La primera, la más obvia, es qué hacer con los desechos electrónicos cuando se acabe la vida útil de la aplicación, y la segunda viene dada por el carácter la comunicación entre los dispositivos, ya que todas las conexiones inalámbricas generan ondas electromagnéticas que pueden afectar a los seres vivos.

9.1 Final de la vida útil de los componentes:

Respecto al impacto medioambiental causado por el hardware como futuro residuo, una vez termina la vida útil del producto que debe ser lo más larga posible, se debe reciclar correctamente para no dañar al medio ambiente. Para ese cometido existen plantas de tratamiento especializadas en residuos electrónicos, ya que el reciclaje de componentes a nivel particular es relativamente inviable.

Los diversos componentes (circuitos impresos, chips etc.) deberían ser desechados de la forma correcta en centros de gestión de residuos de aparatos eléctricos y electrónicos (RAEE). Su desecho en contenedores de basura común supondría su acumulación en vertederos donde el efecto de los materiales contaminantes de la placa (como el PCB, TBBA, PBB, PVC, etc.) resultaría perjudicial para el medio ambiente. En los centros de gestión de residuos se lleva a cabo un reciclaje efectivo de los componentes, se desechan de forma apropiada los elementos contaminantes y finalmente los productos no nocivos son acumulados.

9.2 Radiofrecuencia:

Para evaluar el impacto medioambiental de este proyecto antes de su aplicación práctica es necesario recordar su objetivo último: comunicación fiable y robusta entre el microcontrolador y la plataforma de recepción (en este caso una Raspberry Pi). Ya que la comunicación es inalámbrica, los dispositivos se comunicarán por medio de ondas electromagnéticas. Por este motivo este capítulo se centrará en el posible impacto sobre los seres vivos que estarán alrededor de la aplicación.

La frecuencia de trabajo del XBEE es de 2,4 GHz (enmarcada dentro de la Alta Frecuencia de entre 3 MHz y 3 GHz; y esta a su vez, enmarcada dentro del espectro de radiofrecuencia situada entre los 3Hz y los 300 GHz de todo el espectro electromagnético). Estos espectros de frecuencia, al ser la radiación electromagnética de tipo no ionizante sus efectos biológicos son únicamente de tipo térmico, son los utilizados, por ejemplo, en procesos médicos que buscan la regeneración de tejidos o el calentamiento de ciertas zonas del cuerpo para favorecer su relajación o recuperación. En contraposición a estas aplicaciones se pueden encontrar estudios que indican que los efectos sobre el cuerpo humano ante una exposición durante un largo periodo de tiempo ante una radiación de gran potencia pueden ser: quemaduras, hipertermia, cataratas y esterilidad en el caso de los hombres.

Según la Organización Mundial de la Salud, dentro del rango de frecuencias de entre 1MHz y 10 GHz, en necesario un SAR de por lo menos 4W/Kg para producir efectos adversos en la salud de gravedad. Sin embargo, para un largo periodo de exposición (por ejemplo de un operario que trabaja durante horas), no es admisible una densidad de potencia mayor de 0.4W/Kg para evitar cualquier tipo de lesión. En este caso, ya que los módulos de radiofrecuencia utilizados tienen una potencia de 60mW y suponiendo que la eficiencia de la antena sea el 100% (cosa sumamente improbable) tendríamos una potencia muy inferior a la permitida.

10. Conclusiones

Tras la fase de diseño y verificación del sistema de comunicación por radio frecuencia, se puede concluir que el trabajo de desarrollo responde satisfactoriamente a los objetivos propuestos al inicio de este proyecto.

En primer lugar, se ha conseguido establecer un canal de comunicación inalámbrico entre un microcontrolador y un computador. Mediante la configuración de los módulos XBee, se ha realizado un intercambio de información a diferentes velocidades de transmisión, permitiendo hacer un pequeño estudio del comportamiento de estos módulos. Gracias a la verificación por “checksum”, el sistema es más fiable y evita posibles errores en la lectura e interpretación de las tramas enviadas.

Por otro lado, se ha logrado complementar la información recibida, de manera inalámbrica, con una interfaz gráfica. De esta manera, permite al usuario tener una idea en tiempo real de la orientación del objeto, aún que no lo esté viendo directamente.

En conclusión, esta aplicación proporciona una sólida base sobre la que construir sistemas más complejos, ya que aporta una idea general del funcionamiento de los módulos XBee Pro 60mW - Series 1 (802.15.4). Este proyecto también muestra un una amplia visión del microcontrolador empleado (Pyboard con Micro Python), debido al uso de diversos periféricos, demostrando una gran versatilidad. Juntamente con la Raspberry Pi, la Pyboard y los módulos XBee constituyen un buen paquete de hardware con los que trabajar, gracias a su simplicidad pero sus grandes prestaciones y su precio asequible.

Como recomendación final, simplemente cabe decir que para comunicaciones efectivas y a gran distancia, el módulo de radio frecuencia empleado no sería el más indicado. Aún que con una antena apropiada podría llegar a 1,5 kilómetros (apto para

usos recreativos a media distancia), existen diferentes alternativas (incluso de la misma compañía Digi) que proporcionan mayor alcance y fiabilidad. De todos modos, la aplicación desarrollada en este proyecto solo se contempla para uso privado, y se debería descartar su aplicación en un campo industrial, donde las prestaciones necesarias son superiores. Por ello se recomendaría explorar otras opciones, y que puedan ofrecer garantías suficientes para el uso masivo y sin riesgos para la gente.

11. Agradecimientos

Primero de todo, agradecer a toda mi familia el apoyo prestado durante estos últimos meses, ha sido clave para poder realizar el proyecto satisfactoriamente. También a mis amigos y compañeros que me han ayudado, aportando ideas claves y gracias a sus puntos de vista, he podido superar problemas y algunas encrucijadas con las que me he encontrado.

Por supuesto, también agradecer la enorme paciencia de mi director del trabajo, Juan Manuel Moreno Eguílaz, que con su tiempo y consejo ha encaminado el desarrollo de esta aplicación. También remarcar que, a pesar de ser un tema desconocido para él, siempre se ha mostrado colaborativo e interesado en el proyecto, proponiendo caminos alternativos y dando consejo clave.

Por último, toda la gente que no he tenido el placer de conocer, pero que gracias a su trabajo han hecho posible la finalización de este proyecto.

12. Bibliografía

1) Página oficial de MicroPython [https://micropython.org/]
Consultada en julio 2015

2) Referencias y primeros pasos tutoriales [http://docs.micropython.org/en/latest/pyboard/quickref.html]
Consultada en julio 2015

3) Página oficial de la librería pySerial [http://pyserial.sourceforge.net/]
Consultada en julio 2015

4) Foro online de informática Stack Sverflow [http://stackoverflow.com/]
Consultada en julio 2015

5) Página oficial Digi electronics [http://www.digi.com/]
Consultada junio 2015

6) Tutorial para XBee [http://xbee.cl/tutorial-xbee/]
Consultada julio 2015

7) Repositorio web de código libre GitHub [https://github.com/]
Consultada julio 2015

8) Página oficial Raspberry pi [https://www.raspberrypi.org/]
Consultada junio 2015

9) Página oficial Pygame [https://www.pygame.org/docs]
Consultada junio 2015

10) Review sobre MicroPython (YouTube) [https://www.youtube.com/watch?v=5LbgyDmRu9s]
Consultada junio 2015

11) Tutorial YouTube sobre Raspberry Pi [https://www.youtube.com/watch?v=PgiNmQrMxBc]
Consultada junio 2015

12) Página oficial de Python [https://www.python.org/]
Consultada julio 2015

13) Tutorial de Sparkfun Electronics [https://learn.sparkfun.com/tutorials/exploring-xbees-and-xctu]
Consultada junio 2015

14) Tutorial de YouTube sobre pines GPIO de la Raspberry Pi [https://www.youtube.com/watch?v=PgiNmQrMxBc]
Consultada julio 2015

15) Tutorial de YouTube sobre la configuración XBee [https://www.youtube.com/watch?v=ZK5JC4WpLKQ]
Consultada julio 2015

16) Tutorial sobre el modo API XBee [http://fuenteabierta.teubi.co/2014/03/arduino-y-el-xbee-series-1-modo-api.html]
Consultada julio 2015

17) Artículo sobre MicroPython [http://www.wired.co.uk/news/archive/2013-12/06/micro-python]
Consultada junio 2015

18) Fuente de los archivos del M.I.T sobre la IMU 9250 [https://github.com/micropython-IMU/micropython-mpu9x50]
Consultada julio 2015

19) Página oficial Raspbian [https://www.raspbian.org/]
Consultada junio 2015